Abstract

The Lifestreams Software Architecture

Eric Thomas Freeman
Yale University
May 1997

"Typical" computer users struggle to organize and find their own electronic documents, manage their schedules and correspondence, and filter an ever increasing deluge of information. The process is made worse as users are forced to combine the disparate features of many applications to achieve these tasks. These problems suggest that our current software systems are ill-equipped to handle the demands of the typical computer user. Research has shown that common desktop environments (such as the Macintosh "desktop") are often badly fitted to users' needs.

In an attempt to do better we have reduced "information management" to a few simple and unifying concepts and created "Lifestreams." Lifestreams is a software architecture based on a simple data structure, a *time-ordered stream* of documents, that can be manipulated with a small number of powerful operators to locate, organize, summarize and monitor information.

In this dissertation we first provide motivation for Lifestreams. We then present the model and discuss the development of our research prototype. Our prototype realizes many of the system's defining features and has allowed us to experiment with the model's key ideas with actual users (of differing levels of computer experience) over the course of its development. Results from its use suggest that Lifestreams is an effective software architecture for managing common computer tasks; its simple organizational storage system (the stream) combined with a small number of powerful operators provides a unified framework that subsumes many separate desktop applications to accomplish and handle the most common personal communication, reminding, and storage and retrieval tasks. In addition, Lifestreams suggests valuable new capabilities for electronic systems.

The Lifestreams Software Architecture

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Eric Thomas Freeman
May 1997

Acknowledgements

This dissertation could not have been completed without the support and contributions of my colleagues and friends. First and foremost I owe thanks to my thesis committee: my advisor, David Gelernter, provided inspiration, endless ideas and encouragement throughout my graduate career. Nick Carriero acted as a co-advisor and through technical discussions and his careful reading of the dissertation greatly improved the quality of the final work. Ben Bederson, my external reader, provided outside encouragement and criticism and was particularly helpful with his guidance in the evaluation aspects of this dissertation. Last, Martin Schultz was supportive and interested in the project throughout.

I also owe sincere gratitude to Scott Fertig who acted as a mentor and made many contributions to the work (as well as being a good friend). My wife, Elisabeth, not only provided love and support, but also endured using the early Lifestreams systems and was always the first to read rough dissertation drafts. David Kaminsky interested me in coming to Yale and acted as a mentor early on. Susanne Hupfer and Rob Bjornson were friends and colleagues and filled many days with interesting discussion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Despite rapid advances in many areas of computer science (user interfaces, communications technologies, object-oriented systems) over the last several decades, computers are widely acknowledged to be harder to use than they could or should be [Kap91, MH92, Hil95, Cor96]. Typical computer users struggle to organize and find their electronic documents, manage their schedules and correspondence and filter an ever increasing deluge of data from the Internet. Matters get worse when users are forced to draw on the disparate features of many applications to achieve these tasks.

These problems suggest that our current "operating system interfaces" are ill-equipped to handle the demands of the "typical" computer user. By operating system interface we are referring to the predominant computer interfaces and systems used by typical computer users; namely, those based on the desktop metaphor. Throughout the dissertation we will refer to these systems generically as "current software systems," "desktop systems" and "operating system interfaces." We will also define "typical users" later in this chapter.

The Lifestreams software architecture is an attempt to do better than current systems. Lifestreams is a new software architecture for managing personal electronic information based on a simple data structure — a *time-ordered stream* of documents — combined with a small number of powerful operators for locating, organizing, summarizing and monitoring information. Our prototype implementation (which we present as a "proof of concept") realizes many of the system's defining features and has allowed us to experiment with the model's key ideas.

This dissertation gives motivation for Lifestreams, presents a formal Lifestreams model, and reports on the development of (and our practical experiences with) the research prototype. Our results show that Lifestreams is an effective software architecture for managing common computer tasks; its simple organizational storage system (the stream) and powerful operators provide a unified framework that subsumes many separate desktop applications to accomplish and handle the most common personal

communication, reminding, and storage and retrieval tasks.[1]

1.1 Problems with Current Software Systems

Today computer users predominantly use software systems based on the "desktop metaphor," which attempts to simplify common file operations by presenting them in the familiar language of the paper-based world (paper documents as files, folders as directories, the trashcan for deletion). While this metaphor has been successful to a point (granted one usually has to explain to a new user how the computer desktop is like a real desktop), as we shall see, the paper-based model is a rather poor basis for managing electronic information.

Previous work has considered the ways people use information and electronic systems and identified a number of problems [Mal83, Lan88b, Car82, Coo95, DL83, JD83, Col82, MSW92, Nel90, BN95, Eri96, WS96, Pem96, Kay90]. Users have trouble filing, finding and organizing information; incorporating "reminding" into their electronic environment; archiving and making use of archived information; obtaining "overviews" or summaries of information; managing many separate, incompatible applications and data formats needed to achieve common, everyday tasks. These problems point out where desktop systems fall short of user's needs and suggest directions for improving them. Let's examine each point.

1.1.1 Finding and Filing

Lansdale [Lan88b] has studied the processes of recall, recognition and categorization in an attempt to propose software frameworks that have a basis in psychological theory. He describes information categorization as the problem that lies in "deciding which categorizations to use, and in remembering later exactly what label was assigned to a categorization." Much of Lansdale's work builds on Tom Malone's seminal study of the way people use information: *How Do People Organize Their Desks? Implications for the Design of Office Information Systems* [Mal83]. In his study Malone aimed to obtain a "systematic understanding of how people actually use their desks and how they organize their personal information environments" in an attempt to improve electronic systems. Malone's work suggested that categorizing information is perhaps the most difficult information management task people encounter.

What makes categorizing information so hard and why is information so hard to find after it is categorized? Lansdale found that "quite simply, information does not fall into neat categorizations that can be implemented on a system by using simple labels." The work of Dumais and Landauer [DL83] identified two specific reasons: (1) information falls into overlapping and fuzzy categories and (2) it is impossible for users to generate categories that remain unambiguous over time. Both of these reasons are

[1] The *Thesis*.

problematic for electronic file and folder systems because they cannot represent the true relationships between the information they store. Hierarchies organize documents into strict relationships, but most documents are related to other documents in fluid, multi-dimensional ways. The following examples are suggestive (from [Lan88b]):

> "My boss wants to see all the project reviews I have carried out over the last six months. The trouble is, they are filed under each of the individual projects. It will take me ages to work through and dig them all out."

> "Yes I remember that paper. It came at the same time as the product audit. I can't remember what happened to it, though."

Whittaker and Sidner [WS96] give more examples from user interviews:

> "I don't know where to put it. And.. by making the wrong decision, I could really forget about it."

> "I wish I viewed creating a category as a lightweight activity. And for some reason I don't ... it seems like, you know the more of them I create, the harder it is to find any of them that are there."

> "any piece of information longer than five lines has at least several axes along which you might want to look it up and it really depends on how you're coming at it and what you're thinking about at the time. [Filing] isn't reliable."

These examples suggest that users often file information in a manner that is unhelpful when it needs to be retrieved. The problem occurs because, over time, the way we use information changes. We often want to retrieve information in a context that is different from the one in which it was created.

Retrieving information within a traditional file system amounts to remembering the context of use at the time of creation[2]. This approach scales badly. Once the number of files within a personal file space approaches on the order of a few hundred, the user can no longer remember file locations and is forced to use a "find" application (if one exists) to search for information. Whittaker and Sidner provide another good example from a user interview:

> "So what happened was that size of chunks associated with the categories got large. So now one key stroke would get me to a hundred things. So I really was no better off (filing information)."

[2]This approach has been called "location-based finding" since it involves remembering the "place" in the desktop file system where the information was stored [BN95].

The difficulty of maintaining information over time has bad consequences: users often throw away old information as a coping strategy or, as Lansdale observed, spend little time filing information because there is no immediate pay-back. In either case, information is either "lost" because it was thrown away or difficult to find in an unorganized clutter. On the whole there is a "negative feedback loop:" users have difficulty categorizing information which, in turn, defeats retrieval. Likewise, electronic desktop systems do not aid in retrieval so users are not likely to maintain disciplined filing.

Users are also forced to categorize information in another subtle way: by means of filenames. Lansdale's work has shown that names are an ineffective means of categorizing information. While names act as a mnemonic device, over time their value decays. Carroll [Car82] has studied the use of naming and found that within a small period of time, people's use of file names results in inconsistent patterns which lead to retrieval difficulties.

The difficulty of categorizing information and the lack of rewards for doing so typically leads users (as Malone discovered) not to file information at all[3] in order to overcome "the difficulty of making a decision between a number of evils, and avoid the consequences of having made it."[4] Lansdale also points out (based on empirical evidence) that "humans are not good at categorizing information" and that requiring users to do so is a "flawed psychological process." Whittaker and Sidner found that "folders may be of little use for retrieval." Cole [Col82] found that "users prefer to spend as little time as possible actually filing" and concludes that "the less time spent filing the better."

1.1.2 Reminding

Electronic "reminding" is the use of the computer to aid a user in remembering tasks or events. In the real world we use a variety of artifacts — calendars, PostIt^{TM} notes, day planners — help us remember. In 1983 Malone pointed out the importance of reminding in our paper-based office systems and suggests their inclusion in computer-based systems [Mal83]. Yet desktop systems still provide little support for reminding; while a number of time management, scheduling, and "todo" list applications have come to market, they don't represent a general solution to provide users with this basic capability.

In more recent work, Barreau and Nardi [BN95] observed that desktop computer users often use a file's location on the desktop as a critical reminding function. For instance, at the end of the day a Macintosh user may leave files on his desktop as a reminder of work to be done the next morning. Other users leave electronic mail messages in their in-boxes [WS96]. Lansdale found this behavior "largely idiosyncratic"; we find the use of location-based storage unsatisfying and an easily undermined method of

[3] Rather than categorizing and filing the information they often prefer to place all files into a "top-level" directory, maintain them within mail applications or to delete them.

[4] This is Lansdale's paraphrase [Lan88b] of Malone.

creating reminders. The desktop metaphor has no semantic notion of reminding; such use by users amounts to a coping strategy for lack of anything better. Location-based reminding is an *ad hoc* user convention and its problems are obvious: there is no way to insure that a reminder actually reminds you. Whittaker and Sidner also suggest that the need for reminding is another deterrent to filing, as "filing information means it is less available to *remind* users about that topic" (their emphasis). Malone suggested in 1983 that electronic systems should help in reminding and improve over physical-world systems: "Failing to support [reminding] may seriously impair the usefulness of electronic office systems and explicitly facilitating it may provide an important advantage for automated office systems over their non-automated predecessors."

1.1.3 Archiving

Old information is *generally* less valuable than new, but regularly situations occur in which old information is essential. Everyone can recall times when he needed information he threw away a month ago. Unfortunately current software systems do not make it easy to archive personal information. They give the users no good means of storing old information and no convenient method for retrieving information. As a result, users are left to invent their own schemes or use third-party applications, both of which still fall short with respect to retrieval. Whittaker and Sidner [WS96] quote one user describing his difficulties:

> "I'm reluctant to archive junk ... I know that the consequence of archiving junk is to make it that much harder to find good stuff ...

The problem is that it is difficult to decide what the "good stuff" is *a priori*. The underlying problem with current software systems is that location-based storage and archiving are conflicting goals. Location-based storage assumes a small information collection (basically what the user can remember) and does not scale to large collections of information, as Jones and Dumais [JD83] found in their study of the spatial metaphor. They concluded that a "strictly spatially-based information retrieval system will be particularly vulnerable to increases in the size of database." In fact, they found that subject recall over time was especially poor and indicated that the location-based approach "is simply more vulnerable to general factors of decay associated with the passage of time" than other symbolic means of retrieval. Location-based storage is particularly ill-suited to retrieval because (as we have discussed) information is not always needed in the same way it was originally.

As we have seen, in the end most users remove old information rather than be forced to deal with the implications of storing it or inventing archiving schemes [Eri91, BN95]. In a study of the practices of several groups of "information users," Erickson found that users often "discarded all but the most important information; space constraints, as well as the difficulty of deciding which file folder was most appropriate, deterred

them from saving more." He also found that there was a general feeling among users that the "fewer items saved, the easier it was to relocate them." This is unfortunate because users throw away information that may prove useful (or even critical [Coo95]) because of flaws in software technology — when memory capacity of storage devices is growing. Weiser [Wei91] has suggested that over the next decade the growth of storage technology will make "deleting old files virtually unnecessary" and should allow "radically different strategies of information management."

1.1.4 Summarizing Information

Summarizing is a vital information processing task. Summaries function as an abbreviated form of a document or collection of documents and reduce the amount of information a user must process [KM96]. They allow users to "gain access to and control the flood of information" and in the end "summaries save time" [Hut95].

Summarizing information is nothing new — yet today there are few electronic systems that support automatic summarization. Current desktop systems provide no general purpose support for summaries; they leave the job to special purpose applications. We believe this lack of support has occurred, in part, because of the current, narrow application-centric view of desktop computing—work has focused on developing tools within applications rather than on "globally" improving users access to information at a systems level.

Summaries are available to users through special purpose products such as Intuit's Quicken, which allows the creation of overviews for financial information. But users need summaries for more routine purposes too. Pemberton, as the chairman of a recent ACM workshop on the future of electronic mail [Pem96], cited the need for summarization in electronic mail systems.

In order to manage large collections of personal information, users need quick and simple methods of distilling them into summaries or overviews. Summarizing is important in many contexts and is central to all communication [Hut95]. Building a "flexible summarizing capability into our systems for these purposes will enhance their performance" [Ove95].

1.1.5 Incompatible and Disparate Applications

Computer users must deal with a far greater quantity and range of electronic objects in their work and personal lives than ever before. Doing so requires an ever increasing number of separate applications, even for the most common computer tasks. Nelson [Nel90] has called this task of making combinations of programs fit together "hopelessly arcane" and says that it results in users "having too many separate, unrelated things to know and understand."

Consider a user who manages his email, schedule, contact list and daily writings on a computer. He may have to use four separate applications as well as the underlying

desktop file and folder system. Products such as Claris Works and Microsoft Office aim to improve the situation, however, on closer examination, these products do little to unify the user's electronic environment because they simply bundle (typically four) applications into a common package. While the user can usually cut and paste between applications, these packages still rely on the underlying file system and do not provide true integration. Systems such as the Newton are a step in the right direction, providing a common store and thus improved "compatibility" among applications, yet the resident applications are still remnants of the desktop systems (see Chapter 8).

1.2 Typical Users and Common Tasks

Before suggesting new directions for software systems it is important to know for whom these systems are intended and what kinds of tasks these users undertake. Thus far we have loosely used the terms "typical users" and "common tasks." What do they mean?

Ed Hilpert, in a recent IBM Personal Systems article [Hil95], has suggested that before computers can become a "ubiquitous consumer-level product," they must be made "easier for the broad range of consumers" who use them. According to Hilpert, this group includes a large group of users who have no computer training, who are possibly afraid of computers and who are not inclined to read manuals or attend training classes. For the purposes of this dissertation, we define "typical" or "common" computer users the same way as Hilpert; in general, we believe this class of users includes people who have little or no technical computer training and yet use a computer regularly. Today this segment represents the majority of computer users, yet software systems fail to provide them effective information environments. Lifestreams is designed to address the needs of these users. We also believe the utility of Lifestreams extends further than this particular class of users. Our long-term goal is to improve "computing" and information management for nearly all users, trained or untrained.

By "common tasks" we are referring to the everyday activities of the typical user. To amass a broad collection of common tasks we turned to three sources: previous work in the information management community that has classified tasks and information usage patterns [WS96, Eri96, Eri91, Mal83, Lan88b, Col82, BN95, Pem96, Hal], related systems that target the same class of users (many are described in Chapter 8), and our experience and that of our colleagues. While this latter source of tasks is somewhat biased given that we are advanced computer users (not representative of the norm), we broaden the scope of study by including it. Let us briefly look at these three.

First, the information management community has analyzed and produced classifications for the tasks that typical users perform and the types of information users process. These classifications are helpful because they condense "information management" into a few primary functions and "information" into a few primary types. We will cover these classifications in depth in Chapter 7. However, to summarize briefly: Users by and large perform the following tasks: "working tasks," which include filing

and finding, task management (such as todo lists) and reminding; "personal archiving," which includes organization and categorization of long-term information; and "communication," which includes interaction between users over space and time and may be "one-shot" or an ongoing conversational thread. In addition, users spend a great deal of time processing documents — creating new documents that are stored in a work area, creating documents based on existing documents, or creating documents based on multiple sources of information (as in Lansdale's multi-project example).

Second, we take into account the tasks provided by related systems. For instance, Microsoft's "Bob" was created with the intention of making computing "easy" [Aza95] and includes seven common functions—a letter writer, a calendar, an email application, an alarm clock, and address book, a checkbook and a game.[5] Moreover, we incorporate the many tasks provided by systems covered in Chapter 8 such as personal todo-list managers, contact managers, personal digital assistants (such as the Newton), hotlist managers, and calendar applications. We also include the tasks suggested by prototypes such as Apple's "Knowledge Navigator," which presents a software prototype that supports a "natural style" of computing.

Last, we include tasks from our local computing environment. We perform many of the same tasks as typical users. We will point out some of the differences between our use and that of novice users and present anecdotes from our local system use in Chapter 5.

Now that we have a basis for the users and tasks we wish to support we proceed to suggest new directions for software systems.

1.3 New Directions for Software Systems

Many of the problems highlighted in Section 1.1 point to areas where desktop systems don't match the flexibility of paper-based systems (naming). Others suggest areas where our software systems can do better (reminding). In either case, we need new systems that improve the user's information handling methods. Our proposal is to leave the paper-based models behind and invent new ones. We are not the first to suggest this approach. Early on, Cole [Col82] suggested that "we must not simply automate existing office practices — in some cases this will perpetuate an existing mess." Lansdale has made a more intriguing argument — although the objectives of two systems may be identical (e.g., information storage and retrieval), the strategies used within one technology aren't necessarily appropriate for a different technology. Jones and Dumais applied this philosophy to desktop systems suggesting that the electronic systems of tomorrow should not be "bound by the same constraints we face in retrieving the tangible objects in our world." In any case, these studies in personal information management provide insight into the ways people manage their electronic

[5] Since games have little to do with personal information management we ignore its inclusion in Bob.

environments and information, and clues as to what people are good and bad at.

Our goal with Lifestreams is to reduce the time users spend managing information while increasing their ability to find and make use of their documents and electronic events. We hope to create a software environment that more naturally supports the ways people work with electronic information and simplifies their electronic interactions.

Based on the results of the studies we have visited in this chapter, we have concluded that the following are essential:

- *Storage should be transparent.*

  "Naming" a file as it is created and choosing a location are unnecessary overhead. When someone starts writing on a piece of paper he doesn't have to give it a name and choose a folder to store it in. Filenames are sometimes pointless and over time often become useless for retrieval purposes; names should only be required when a user explicitly wants one. Likewise, storage locations (in the sense of folders) are effective only as long as the user remembers them; the details of storage should be handled automatically by the system.

  Transparent storage also applies to the network; data should be accessible everywhere, regardless of the viewing device. Users should be able to access their personal information from any available platform—from a Unix machine at work, a Mac or PC at home, a PDA on the road, even a set-top box via cable.

- *Conventional directories are inadequate as an organizing device, information should be organized on demand.*

  Common file systems are too faithful to the paper-based world; paper can't be in more than one place, but electronic documents can (or can behave that way). Desktop systems force users to store new information in fixed categories (namely, directories). But information should be organized as needed, not once and for all when created. Directories should be created *on demand*, and documents should belong to as many of them as seems reasonable, or to none. Directories should be created based on the "semantic" information in the document. This semantic information provides a richer space for users to explore than is provided by the 2D (or 3D) space in most desktop systems, especially over time [Dum96].

- *Computers should make "reminding" convenient and effective.*

  Studies confirms that reminding is a critical function of computer-based systems, yet desktop systems supply little or no support for it. Users are forced either to use location on their graphical desktops as reminding cues or to use add-on applications such as calendar managers. We have argued that the former is a mere coping strategy (for lack of a better method), while the latter could clearly be improved if software systems provided built-in natural support for reminding.

- *"Archiving" should be automatic.*

  Often, users throw out old data rather than undertaking the task of archiving it. In the process they throw out potentially valuable information. Software systems should "automate" archiving, moving old data out of the foreground as it ages, yet keeping it readily available for retrieval.

- *Software systems should be able to summarize a set of documents, yielding a concise overview quickly.*

  With the rapid growth in online electronic data we must all deal with a greater amount of information every day, yet people can't read or process information any faster than they could before [Hut95]. The "full text revolution[6]" implies a "pressing need for automatic summarizing" [Jon], yet conventional user systems do not support a general mechanism for summaries. It is vital that software systems allow users to summarize or compress a large group of related documents into a concise overview quickly; software systems should provide a general purpose facility for summarizing documents, giving users better control over their information collections.

- *Software systems should simplify the user's computer interactions, not complicate them.*

  Performing common computer tasks requires too many separate applications and format translations. Systems should provide an integrated environment for personal data that simplifies and subsumes many applications and the need for explicit importing, exporting or format translation of data.

We add one additional characteristic based on previous work in computer science "systems" research:

- *Software systems should provide the capability for users to customize and create their own extensions.*

  While there is a long tradition of providing extensible systems in the computer science community (UNIX tools, emacs, etc.), human-computer interaction researchers such as Erickson [Eri96] have also discussed the need for flexible environments that can be customized and extended to meet the user's needs.

1.4 Outline

The remainder of this dissertation proceeds as follows: in Chapter 2 we provide a formal description of Lifestreams, which acts as a foundation for the development of our

[6]The phrase "full text revolution" is used by the text summarization community to describe the trend from maintaining only abstracts and keywords (typically in information retrieval systems) of documents online to storing the entire text of the document.

research prototype. The research prototype is described over two chapters: Chapter 3 describes the Lifestreams infrastructure and server, including a discussion of the pragmatic issues involved in adapting our model for "real-world" use; Chapter 4 describes three user interfaces to the Lifestreams system including an X Windows based UNIX workstation client, a command-line interface and an interface that runs on a Newton personal digital assistant. Chapter 5 presents Lifestreams in context, describing how common tasks are accomplished and then demonstrating how Lifestreams is easily extended to support new behaviors. We then present the evaluation of our prototype in Chapter 6. In Chapter 7 we revisit the work of Malone, Lansdale and others from the information management community and examine Lifestreams in the context of their work. Chapter 8 surveys the landscape of related systems and applications, comparing and contrasting each with Lifestreams. Finally chapter 9 presents conclusions and suggests directions for future work.

Chapter 2

The Model

In this section we present the Lifestreams model. We begin with a conceptual description and then describe the model informally in light of the operational characteristics that we determined necessary for such systems in Chapter 1. We then provide a formal description of Lifestreams. The reader may skip the formal section without loss of continuity, but it is recommended for readers who care about implementation details.

2.1 The Concept

Lifestreams have their beginnings in the "chronical streams" of Gelernter in [Gel91] and were first described as a structure for managing personal electronic information in [Gel94]. This dissertation is the first effort to define Lifestreams in concrete terms and to demonstrate its usefulness via a working prototype. In the spirit of the initial writings about Lifestreams, let us first examine Lifestreams as it might look in 1999 (adapted from [CFFG96]):

> Here's your compute-environment circa 1999: every document you've ever created or received stretches before you in a time-ordered stream, reaching from right now backwards to the date you were born. You can sit back and watch new documents arrive: they're plunked down at the head of the stream. You browse the stream by running your cursor down it—touch a document in the display and it pops out far enough for you to glance at its contents. You can go back in time, or go to the future and see what you're supposed to be doing next week or next decade.
>
> Every chunk of information (every document, email message, application transcript, rolodex card, appointment-calendar item...) is stored in a single time-ordered stream. When you tune in, you see a stream of documents receding into the distance; farther away in imaginary space means farther back in time. To create a new document, you can press the "new" button

and get an empty box ready to fill, or clone an old document and get a new copy to alter as you choose. You don't need to name documents (although you can); documents are located by attribute and chronology. When you want to create a new document, you spend zero time deciding where to put it and what to name it.

You might be in danger of being overwhelmed by all the documents on your screen—but you use "substreaming" by pressing "find" and describing the documents you want, and everything else (temporarily) disappears. A substream persists until you kill it. A newly-arriving document gets dumped in the main stream and also appears on every substream where it fits.

When you press the "squish" button you get a summary of a substream. The type of summary depends on the type of information in the substream—textual for plain documents, graphs or pictures or animations for the appropriate more-specialized types. The "squish" button automatically invokes an appropriate *squish* for this substream (or offers you a choice of reasonable squishers). In some cases, highly complex and sophisticated squishers will be desirable. The lifestream system's contribution isn't to say how these squishers should be built—rather to suggest *that* they be built, and to provide a uniform framework in which they can be installed.

The stream has a future as well as a past. Appointments and calendar items are stored in the future, and become visible when their creation-times roll around or when you go to the future on purpose to look around.

Documents in the "present" are writable. Farther back, in the "past," they have frozen into history and become read-only. Each user decides when the present ends and the past begins—at what point, in other words, documents freeze. One possibility (and the system default) is to freeze today's documents at the start of tomorrow. In that case you can work on a document all day, but to continue the next day you need to use "clone" to create a new copy on the streamhead. Or a user might postpone freezing for a week, or forever. The far-tail of the stream—for example, documents that are more than two years old—may disappear at the implementation's discretion into archival storage. The user specifies where the "far tail" begins, but the Lifestreams-provider will presumably set charges that depend on a user's willingness to have old material dumped into data warehouses.

Agents can troll down lifestreams; an agent can cruise to the head of a stream for example and go to sleep after posting instructions that it be awakened whenever a document arrives. Agents are important to many aspects of the system; custom agents are the main ways in which the system is designed to accommodate extensions and refinements.

Now that we have presented the concept of Lifestreams "in the large," we present the model more concretely. We do so in terms of its storage paradigm and its basic operations. In the process we suggest how Lifestreams provides transparent storage, organization through on-demand directories, and the ability to create overviews. We then examine the underlying time-based storage model and, in the process, show how Lifestreams accomplishes archiving and reminding in a natural way.

We should be clear at the onset that the prototype implements the ideas needed to prove the thesis. It is not a complete embodiment of the ideas.

2.1.1 What is a lifestream?

A *lifestream* is a time-ordered stream of documents that functions as a diary of your electronic life; every document you create or other people send you is stored in your lifestream. The tail of your stream contains documents from the past (starting with your electronic birth certificate, perhaps). Moving away from the tail and toward the present, your stream contains more recent documents — papers in progress or new electronic mail; other documents (pictures, correspondence, bills, movies, voice mail, software) are stored in between. Moving beyond the present and into the future, the stream contains documents you *will* need: reminders, calendar items, to-do lists.

Users interact with Lifestreams via five operations: `new`, `clone`, `transfer`, `find` and `summarize` depicted in Figure 2.1. We will examine each operation.

2.1.2 Document Creation and Storage

Users create documents by means of `new` and `clone`. `New` creates a new, empty document (ready for editing) and adds it to your stream; the user may be given the choice between a selection of document types and/or templates (depending on the document model). The user does not explicitly manage the document's storage (in terms of its location) — documents are always added to the head of the stream and don't require names unless the user so desires. Each document comes with a number of attributes (such as its creation date and time), most of which are inferred by the system. Other attributes (such as a document's content type) can be changed by the user if need be.

The `clone` operation creates a duplicate of an existing document, ready for editing, and adds it to the head of your stream. As with `new`, storage is implicit because the duplicate is added to a default "location" (the head of the stream) and naming is not necessary. The document's attributes are inherited from the original document. In many systems, the information in one document often becomes the basis for the creation of another [Hal]. The `clone` operation provides a means of creating a new editable version of an existing frozen document in Lifestreams.

The `transfer` operation creates a copy of an existing document on another lifestream. The document is added to the head of the stream (by default) in a frozen state (to

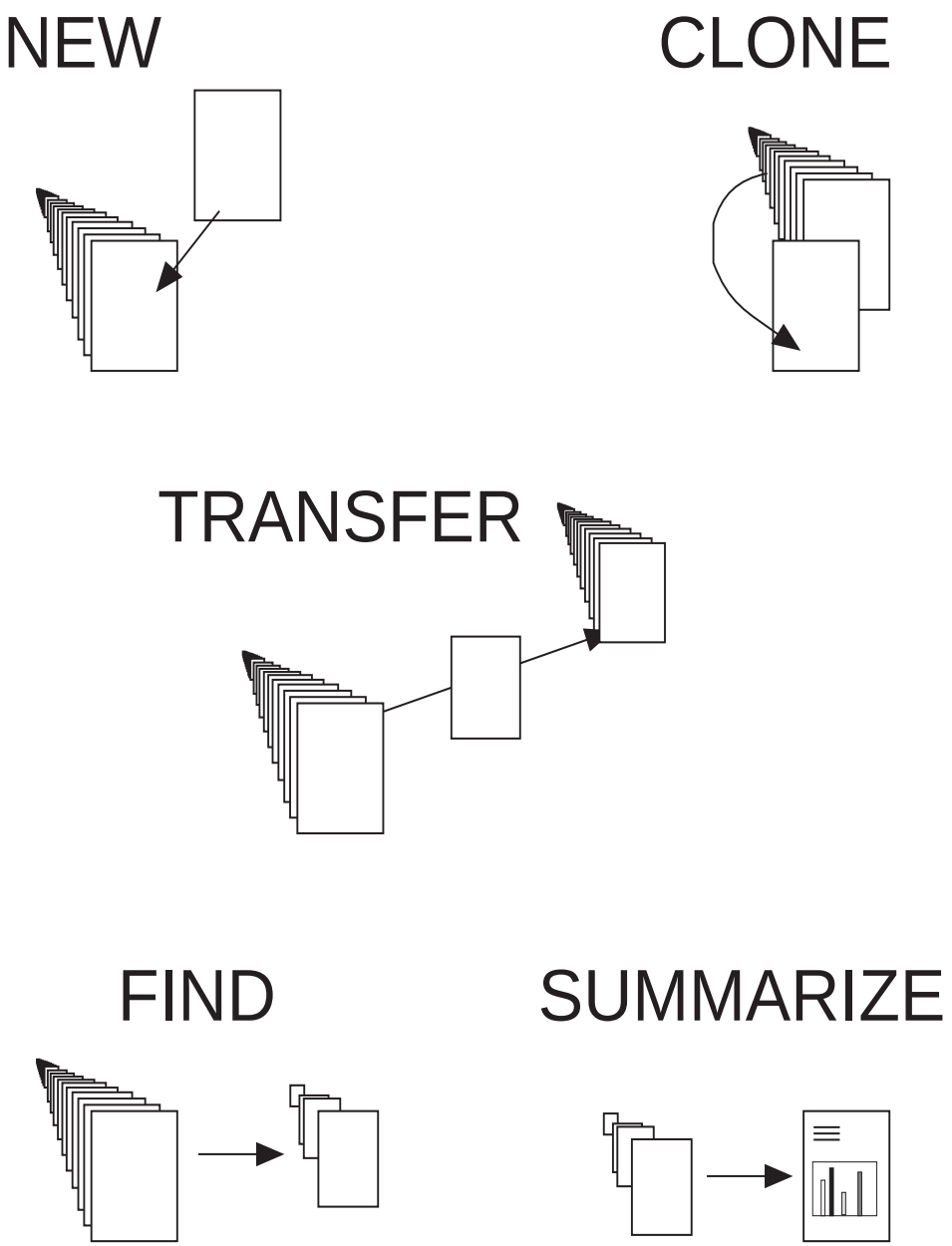


Figure 2.1: The Basic Lifestreams Operations.

preserve the method in which the document was created) and updated to reflect the sender and receiver of the document.

2.1.3 "On Demand" Organization

Lifestreams are organized on demand with the `find` operation. `Find` prompts for a search query, such as "all documents that mention Toaster Ovens," "all letters to Schwartz," "my web bookmarks," applies the query to the user's stream and builds a *substream* that contains only the documents that match the search query.

Substreams, like virtual directories [GJSO91, MW93], present the user with a "view" of a document collection. The view contains all documents that are relevant to the search query. Substreams differ from conventional directory systems in that, rather than placing documents into fixed, rigid directory structures, they create virtual groups of documents. Documents aren't physically stored in them; a substream is a temporary collection of documents that already exist on the main stream. Substreams may overlap and can be created and destroyed on the fly without affecting the main stream or other substreams.

Substreams are dynamic. If you allow one to persist, it will collect new documents that match your search criteria as they arrive from the outside or as you create them. The result is a natural way of monitoring information— the substream acts not only as an organizational device, but as a filter for incoming information. For example, a substream created with the query "find email" would subsume your mailbox (both your inbox and outbox) and automatically collect mail as it arrives.

2.1.4 Overviews

The last operation, `summarize`, compresses a substream into an overview document that is added to the user's stream. The content of the overview document depends on the type of documents in the substream. For instance, if the substream contains the daily closing prices of all the stocks and mutual funds in your investment portfolio, the overview document may be a chart displaying the historical performance of your securities and your net worth. If the substream contains a list of tasks you need to complete, the overview document might display a prioritized "to-do" list. No matter how many documents fall into a given category Lifestreams will summarize them into a single document. Summaries are meant to be a flexible part of the Lifestreams system that can be extended by advanced users to create their own summary types and share with other users.

2.1.5 Agents

Agents are also an integral part of the system. Agents can be attached to documents, streams or the user interface to automate tasks or extend the behavior of Lifestreams. These agents act as "embedded computations" that are executed on various events,

such as the arrival of a new document on a stream. A number of agents come with any Lifestreams system; agents can also be created by advanced users.

2.1.6 Chronology as a Storage Model

We use chronology as a storage model for one reason: time is a natural guide to experience. It is the attribute that comes closest to a universal "road map" for stored experience. Previous studies support this view. Malone suggested the utility of time-based organization in his early studies. Lansdale conducted experiments which showed that "people remember chronological information about information" and that chronology is a powerful memory cue for information retrieval. Erickson [Eri91] found that users often remember the approximate dates of information.

The chronological stream adds historical context to a document collection; all documents eventually become read-only (frozen in the past, set in stone for history), and the stream preserves the order and method (e.g., who created the document, if it was transferred to the stream) of their creation. Freezing a document (that is, making it read-only) can be done manually by the user or automatically by the system (e.g., the system freezes all documents at 3:00 am every night or all documents that haven't been changed in two weeks). Like a diary, a stream records work, correspondence and transactions. Allen [All83] observed that in some applications (such as medical records) the "course of events becomes a critical part of the data." A historical record can also be crucial in organizational settings. Cook [Coo95] has studied the long term "institutional memories" within organizations and found that many of the assumptions inherent in our paper-based systems (e.g, when a user leaves a job his "files" are not likely to leave with him) have not been translated to our electronic systems. Electronic documents, for instance, are easily deleted or misplaced. When an employee leaves the job, his entire data collection may disappear. This problem has proved to be critical in a number of environments. Cook states that the key to maintaining critical electronic information "lies in being able to determine, sometimes long after the fact, not only the content but also the context of a record in question." In essence, this amounts to not only having the data at hand but knowing the lifetime of a piece of data, the "when, by whom, where, how ... over time" of the data.

Chronology also provides a powerful and natural method of adding "reminding" to electronic systems. In reminding, the user relies on some future event to remind him of some task. The future of the stream can be used to store reminders — by placing a document in the future part of the stream, the system can alert the user at the precise "time" of the reminder. We will see in Chapters 4 and 5 some uses of this capability.

2.2 The Formal Model

We now present our formal description of Lifestreams. We do so in an incremental fashion, first supplying a model that supports a general document collection with dynamic

filters. We then refine the model to provide the complete semantics of Lifestreams; namely a system that supports time-based ordering, dynamic incremental filters, and an extendable communication system via agents. Throughout the description we make use of the Linda coordination language [CG89] as a means of providing a "concurrency semantics" for Lifestreams.

2.2.1 The Linda Coordination Language

Linda is a coordination language developed by Nicholas Carriero and David Gelernter [CG89] that compliments "traditional" languages for computation (C, Fortran, Ada, etc.). Merging Linda with these computation languages produces dialects that can be used to write parallel and distributed programs. Linda is also useful in the specification of distributed systems as it provides an underlying coordination semantics.

Linda's coordination model is a simple one: a small number of operations are used to read, write and erase a (logically) shared, synchronizing, associative memory. This memory, called a tuple space, stores tuples that are simply lists of typed fields (where the types come from the base language). Linda's operations fall into two groups, those that generate new information to be put into tuple space (**eval** and **out**) and those that extract data from tuple space (**in** and **rd**).

More specifically, **out** converts its argument list into a tuple and puts that tuple into tuple space. **Eval** spawns a separate thread of control to evaluate each argument. When all threads complete, this "live" tuple resolves into a normal tuple. The **in** operation converts its argument list into a template and then searches tuple space for a matching tuple. If one or more matching tuples exists then **in** removes and returns the first tuple found. Otherwise **in** blocks until a matching tuple is added to the tuple space. The **rd** operation works like **in**, except that it does not remove the matching tuple, rather it returns a copy of it.

The matching rules for tuples and templates are fairly intuitive:

1. The tuple and template must have the same number of fields.
2. Corresponding fields of the tuple and template must have the same type.
3. Corresponding fields that contain data must be equal.
4. Two corresponding fields cannot both be place holders.

We will now define the Lifestreams model using a tuple space to store both streams and documents. In more detail, we represent each document with a tuple and each stream by three types of tuples: one tuple that maintains a count of the number of documents in the stream, a tuple for each document in the stream (that is a reference to a document tuple), and a tuple for each substream.

2.2.2 Documents and Attributes

A *document* is the primitive data type in the Lifestreams model. While an implementation of Lifestreams may assume a specific document model, the Lifestreams model treats the data making up a document as a finite array of bits. This array of bits may represent a text document, an image, a movie, a fax, an application, or anything else that is representable by bits. Note that we could have defined the most primitive data type with a smaller granularity than a document (such as a section or paragraph) but that would have implied a more specific document model and introduced additional complexity (though such refinement could be added to our model if needed).

Each document is represented as a collection of descriptive adjectives in the form of attributes/value pairs. Each pair describes some aspect of the document — the `created` attribute describes the date and time a document was created, the `creator` attribute describes the user who created a document, and the `type` attribute describes a document's content type. The array of bits making up the content of the document is stored in an attribute called `data`. The model does not fully specify a set of such attributes for every Lifestreams implementation because different attribute sets may be appropriate for different domains. Take for example the medical domain, which may need a set of specialized attributes for keeping track of patient records. However, a minimum set of attributes are necessary in all instantiations of the model as they are crucial to the primitive operations. Two examples are the `state` attribute, which describes whether or not a document is mutable, and a `created` attribute, which is used in the ordering of documents as previously described. In general, the values of many attributes can be inferred automatically by Lifestreams, although some values can be user specified. For instance, the user may want to provide a list of keywords that may help identify a document more readily in the future. Throughout the chapter we make use of the attributes in Table 2.1, which are listed along with their type and functionality in the model. We will discuss the attributes used in our implementation in Chapter 3.

We formally define a document as a tuple of the form

```
("document", docid, DS)
```

where `docid` is an integer that uniquely identifies the document and `DS` is a document structure in the form of a record with one field per attribute. We define two procedures that act on document tuples: *extract* and *replace*. Given a document the extract (replace) procedure is used to access (alter) the values of its attributes.

More formally, the procedure *extract* (Figure 2.2) takes a document identifier, `docid`, and an attribute descriptor, `attr`, and returns the value of the attribute. To obtain the value, *extract* first reads the `"document"` tuple that matches `docid` from tuple space and binds `DS` to its document record. The procedure *ds_extract* is then

Attribute	Type	Function
`created`	time	Date and time of creation
`state`	boolean	True if document is writable
`data`	array	Content of document
`type`	string	Content type of document

Table 2.1: The document attributes along with their type and function.

used to extract the attribute, `attr`, from the document record.[1] The procedure *replace* (Figure 2.3) takes a `docid`, an attribute descriptor, `attr`, and a value, `val` (*replace* is a polymorphic procedure accepting `val`s of different types), and alters the value of the respective attribute in `DS`. This is accomplished by **in**ing the document tuple from tuple space, replacing the attribute `attr` in the document record with the value `val` and then **out**ing the tuple back into tuple space.

```
proc extract(int docid, char attr)
begin
    document DS;

    rd ("document", docid, ?  DS);
    return ds_extract(DS, attr);
end
```

Figure 2.2: The definition of extract.

Other procedures can be implemented on top of these basic operations, for instance, the *freeze* procedure (Figure 2.4) can be used to make a document read only. *Freeze* takes a document identifier and calls *replace* to set the value of its `state` attribute to `false` (read-only).

The documents of all streams (we will define streams in the next section) are maintained in the same tuple space. Document identifiers in this space range from zero (the first document created) to $n - 1$, where n is the number of documents in tuple space. The value of the next available document identifier is maintained in a tuple of the form

```
("documenthead", count)
```

[1] The procedure *ds_extract* (*ds_replace*) is a simple macro that extracts an attribute from a document record (replaces the value of an attribute in the document record).

```
proc replace(int docid, char attr, poly val)
begin
    document DS;

    in ("document", docid, ?  DS);
    ds_replace (DS, attr, val);
    out ("document", docid, DS);
end
```

Figure 2.3: The definition of replace.

```
proc freeze(int docid)
begin
    replace(docid, state, false);
end
```

Figure 2.4: The definition of freeze.

where `count` is an integer value that is incremented each time a document is added to a stream.

2.2.3 Streams and Substreams

A *stream* is a chronologically ordered collection of documents that acts as the underlying storage framework for Lifestreams. We define a stream as a collection of tuples along with five procedures *append*, *read*, *write*, *retrieve*, and *filter*. Conceptually, *append* adds a new document D to the head of the stream, *read* returns a copy of a document from a stream, *write* replaces a document D with a new version D', *retrieve* retrieves the list of documents in a stream or substream, and *filter* creates a new grouping of documents (a substream) based on a search query.

Formally, a stream consists of a collection of three types of tuples:

```
("stream", sid, dochead, subhead)
("streamdoc", sid, docindex, docid)
("substream", sid, subid, query)
```

One `"stream"` tuple exists for each stream. This tuple contains an integer stream identifier, `sid`, and maintains two integer values: `dochead` and `subhead`. `Dochead` is used to maintain the number of documents within the stream. `Subhead` is used

to maintain the number of substreams in the stream. One `"streamdoc"` tuple exists for each document in the stream. This tuple type contains a stream identifier, `sid`, a document index, `docindex` (a unique value between 0 and $n-1$, where n is the number of documents in the stream), and the `docid` of the `"document"` tuple that contains the document. The `"substream"` tuple maintains the search queries of substreams. One tuple exists for each substream. The substream tuple contains a stream identifier, an integer substream id, and a search query in the form of a character string.

An example tuple space is shown in Figure 2.5; this tuple space contains two lifestreams (with identifiers 0 and 1). Lifestream 0 has three documents with document indexes of 0, 1 and 2, which correspond to the `"document"` tuples 1, 3 and 4. Lifestream 0 also has one substream with a search query of "toasters and 1930". Note that the `"stream"` tuple for lifestream 0 contains a count of the number of documents and the number of substreams. Likewise, lifestream 1 has two documents that are maintained in `"document"` tuples 0 and 2 and two substreams.

Append (Figure 2.6) takes a stream identifier and a document record `DS`, which contains the attributes of the document to be appended. To append a document, *append* must create a `"document"` tuple that contains the document and a `"streamdoc"` tuple that designates the document as belonging to the stream and "points" to the `"document"` tuple. Before creating the `"document"` tuple, *append* first **in**s the document head tuple, binding `docid` to its second field (which will be used as the new document's id). *Append* then returns the tuple to tuple space, incrementing the document count. Next *append* **in**s the `"stream"` tuple that matches the stream identifier `sid`. The value of the stream head is bound to `docindex` and will be used as the document's index within the stream. *Append* then returns the `"stream"` tuple to tuple space with the stream head incremented by one.

Now *append* can create the `"document` and `"streamdoc"` tuples; *append* first **out**s a `"document"` tuple containing `docid` and the structure `DS`. *Append* then **out**s a `"streamdoc"` tuple that contains the stream id, the document index, and a "pointer" (in the form of `docid`) to the `"document"` tuple. Last, *append* returns the value of the document index.

The *read* primitive (Figure 2.7) takes a stream id, `sid`, a document index, `docindex`, and a document record, `DS`, and returns the document's attributes via the document record. *Read* first **rd**s a `"streamdoc"` tuple (that matches `sid` and `docindex`) to retrieve the value of its corresponding `docid`. *Read* then **rd**s the `"document"` tuple matching `docid` and binds its document record to `DS`. The document structure `DS` is then returned.

The *write* primitive (Figure 2.8) (like *read*) takes a stream id, a document index and a document record and writes the document record back to tuple space (in effect, replacing a document with a new one). Like the *read* primitive, *write* first reads a `"streamdoc"` tuple to retrieve the value of its corresponding `docid`. *Write* then extracts the value of the document's `state` attribute to ensure that it is a writable document. If so, then *write* **in**s the `"document"` tuple matching `docid` and then **out**s

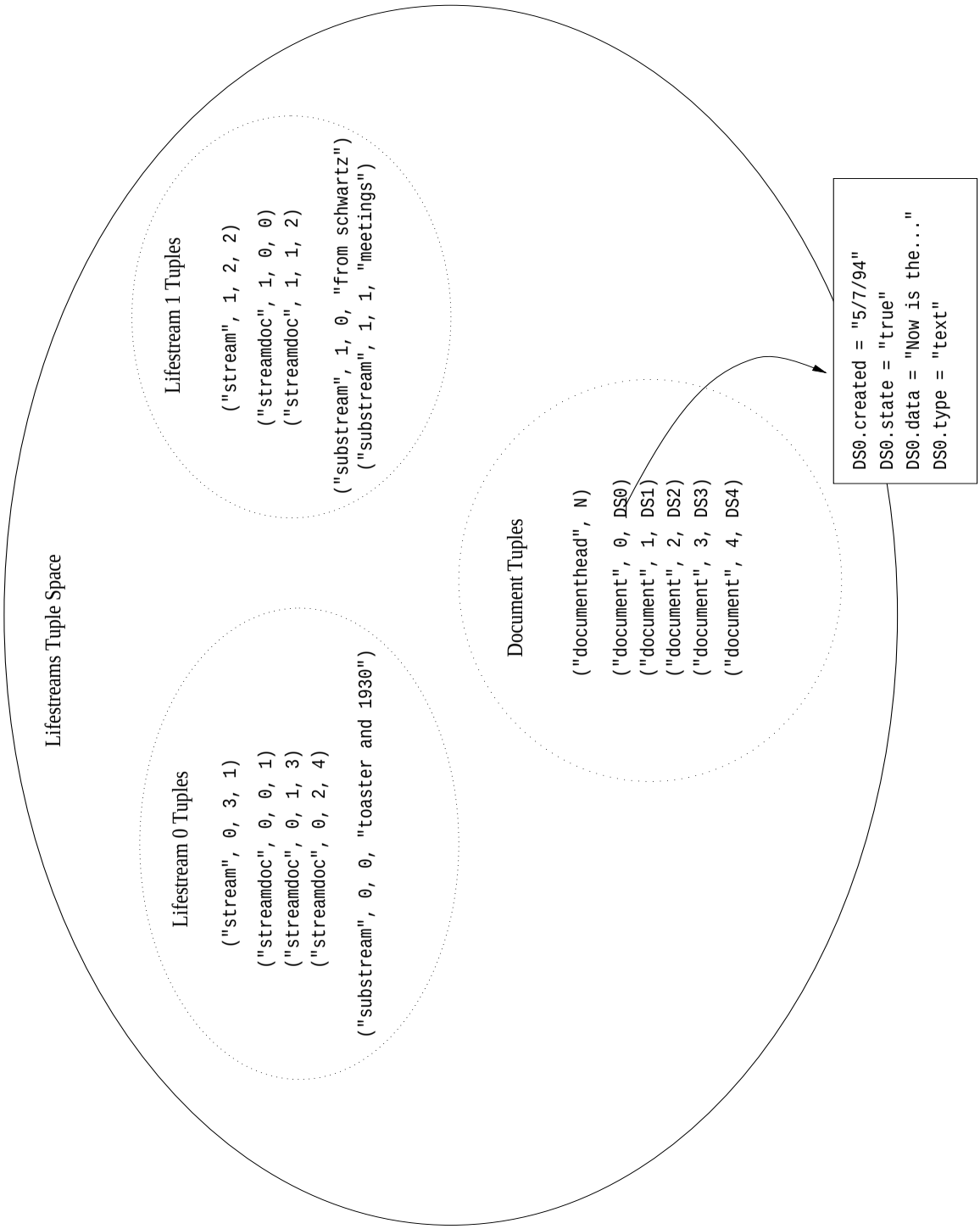


Figure 2.5: Lifestreams Tuple Space.

```
proc append(int sid, document DS)
begin
    int docid, docindex;

    in("documenthead", ? docid);
    out("documenthead", docid + 1);
    in("stream", sid, ? docindex, ? subhead);
    out("stream", sid, docindex + 1, subhead);
    out("document", docid, DS);
    out("streamdoc", sid, docindex, docid);
    return docindex;
end
```

Figure 2.6: The definition of append.

```
proc read(int sid, int docindex, document DS)
begin
    int docid;

    rd("streamdoc", sid, docindex, ? docid);
    rd("document", docid, ? DS);
    return DS;
end
```

Figure 2.7: The definition of read.

the same tuple, replacing the old document structure with `DS`.

The *filter* primitive (Figure 2.9) takes a stream id, `sid`, and a character string search query, `query`, and creates a substream. *Filter* first **in**s the `"stream"` tuple to obtain the value of the substream head (bound here to `subnum`). The tuple is then returned to tuple space with the value of the substream head incremented by one. *Filter* then **out**s a new `"substream"` tuple containing the stream id, a substream id, and the search query. This tuple associates the new substream with the stream. Last, *filter* returns the substream identifier of the newly created substream. The reader will note that we do nothing to actually generate the substream in the *filter* primitive, we merely stash away the search query itself for later use. We will see how search queries are applied shortly.

The *retrieve* primitive in Figure 2.10, takes a stream id, `sid`, and a substream id,

```
proc write(int sid, int docindex, document DS)
begin
    document oldDS;
    int valid;

    rd("streamdoc", sid, docindex, ? docid);
    valid := (extract(sid, docid, "state") = writable)
    if (valid) then
    begin
      in("document", docid, ? oldDS);
      out("document", docid, DS);
    end;
end
```

Figure 2.8: The definition of write.

`subid`, and returns a list of documents (the substream). To create this list, *retrieve* first **rd**s the `"substream"` tuple and retrieves the `query`. *Retrieve* then reads the `"stream"` tuple (of the substream) to obtain the document count (bound to `docnum`) of the entire stream. Next, *retrieve* creates a list of the document ids in the stream by calling **iota**[2] with the number of documents in the stream. *Retrieve* then steps through each document in the list and calls *match* with its stream id, document id and the query. If the document is a "match" then it is cons'd onto the `substream` list. Last, *retrieve* sorts the resulting list by the creation of the documents and returns the list. We leave both *sort* and *match* undefined[3].

As we mentioned our model generates substreams from queries when *retrieve* is called rather than when the substream is created. For the purposes of our model this gives us a clean way of expressing the *dynamic* nature of substreams. Substreams are dynamic in the sense that they continue to collect documents as new documents are added to the stream collection. A substream created from a query that matches "all documents about X" will continue to collect documents referring to X as they are added to the stream after the substream's creation (because the query is applied each time a substream is retrieved).

While this provides a clean model we will see that in implementing Lifestreams we need to take an incremental approach to creating substreams. This approach is consistent with our semantics, yet still satisfies our real-world need for efficiency. We

[2]Given an integer n, **iota** generates a list $(0, 1, 2, \ldots, n-1)$.

[3]Various document and retrieval models will call for different definitions, but the intended use should be obvious.

```
proc filter(int sid, char query)
begin
    int docnum, subnum;

    in("stream", sid, ? docnum, ? subnum);
    out("stream", sid, docnum, subnum + 1);
    out("substream", sid, subnum, query);
    return subnum;
end
```

Figure 2.9: The definition of filter.

will return to this point in the next chapter.

2.2.4 Lifestreams Refinement

We have now defined a general document model that defines the basic functionality needed for Lifestreams, namely the ability to append documents to a stream, read and mutate documents, and to create substreams through filters. We are still missing a means of creating a chronologically-ordered default stream, the ability to incrementally create substreams, and the ability to extend the system via agents. We now further refine the model to include these capabilities.

Incremental Substreams

Substreaming (through *filter* and *retrieve*) can only be applied to the stream itself; that is, substreams are generated by filtering the entire collection of documents in a stream and cannot be created by filtering another substream. It would be convenient and useful to be able to apply queries to substreams. This allows for an incremental approach to creating substreams: for instance, one might create a substream to filter all documents about "investments" and then create another substream from the investment substream that filters out documents about "mutal funds." This approach has some nice properties with respect to the implementation as we will see in chapter 3. It also allows for the organization of substreams into hierarchies of sorts (note however that this is not a true hierarchy—substreams can contain overlapping sets of documents).

We now extend our definition of *filter*, *retrieve* and the `"substream"` tuple type to support incremental substreams. The transformation is straightforward: to *filter* (Figure 2.11) we first add an extra parameter: the "parent" substream id, `pid`, of the source substream. We then alter the **out** on the `"substream"` tuple so that it also

```
proc retrieve(int sid, int subid)
begin
    string query;
    int docnum, subnum, doc;
    list docs, substream := ();

    rd("substream", sid, subid, ? query);
    rd("stream", sid, ? docnum, ? subnum);
    docs := iota(docnum);
    foreach doc in docs
       if (match(sid, doc, query))
         substream := cons (doc, substream);
    return sort(sid, substream, "created");
end
```

Figure 2.10: The definition of retrieve.

contains an additional field (the parent substream id).

To *retrieve* (Figure 2.12) we first update the initial **rd** to match on the extra `pid` field in the `"substream"` tuple. We then alter the manner in which we create the list of documents for *match* to iterate through — if the parent substream of this substream is `0` then we iterate through all documents in the lifestream as before by using **iota** to enumerate all documents between `0` and `docnum - 1`. Otherwise we recursively call *retrieve* to create the list by retrieving the parent substream. In this way a substream is generated by first applying the query of the "parent" substream (and any queries it relies on) and then applying the current query.

```
proc filter(int sid, int pid, char query)
begin
    int docnum, subnum;
    in("stream", sid, ? docnum, ? subnum);
    out("stream", sid, docnum, subnum + 1);
    out("substream", sid, subnum, pid, query);
    return subnum;
end
```

Figure 2.11: The definition of incremental filter.

```
proc retrieve(int sid, int subid)
begin
     char query;
     int docnum, subnum, doc, pid;
     list docs, substream := ();

     rd("substream", sid, subid, ? pid, ? query);
     rd("stream", sid, ? docnum, ? subnum);
     if (pid == 0)
       docs := iota(docnum);
     else
       docs := retrieve(sid, pid);
     foreach doc in docs
        if (match(sid, doc, query))
          substream := cons (substream, doc);
     return sort(sid, substream, "created");
end
```

Figure 2.12: The definition of incremental retrieve.

The Default Chronological Stream

Our model now gives us a general framework for describing a collection of documents and an arbitrary number of chronologically-ordered subsets of those documents (they are chronological because we always sort them by their creation date). In Lifestreams we would like to further refine the model to include a default substream (the lifestream) that contains all documents within the document collection. We specify the default substream (for the `sid` = 0) as:

$$\texttt{lifestreams} := filter(0, 0, \texttt{"*"});$$

where `"*"` is a wildcard that matches any document in the collection. We will describe our specific filtering mechanism in Chapter 3.

Handling Time Correctly

When *retrieve* is called, it orders the resulting substream based on its `created` attribute. The *append* primitive does not set the value of the `created` attribute when it adds a new document to a stream (the user may have wanted the document to have a future time); in other words *append* "trusts" that the value has been set correctly by the creator of the document. The implication of these two observations is that the

creator of the document is in control of where in the default substream the document is placed. Before we further explore this topic, let's first discuss the intended role of time, and the `created` attribute in Lifestreams.

Lifestreams treats documents differently depending on their creation time: documents in the past are considered "frozen" and (like history) unchangeable. Documents in the present and future are mutable and can be altered. The same is true of document creation: it is forbidden in the past (to allow this would undermine the historical context of the stream as the past could always be altered) and allowed in the present and future. The state of a document — that is, whether it is read-only or writable — is a related issue. Each document in a stream is either writable or read-only as defined by its `state`. We make no specific rules in the model to enforce this behavior, but we assume that all instantiations of the model will adopt some policy such that over time all documents eventually become read-only (or rather, move from the present to the past).

Given these semantics two alterations are needed to correctly support time: one to *append* and one to *write*. In the case of *append* (Figure 2.13) we now only append a document if its creation date is greater than or equal to `now`[4]. We define `now` to be the current clock time[5], and do not further specify how this value is obtained in the model. In the case of *write* (Figure 2.14) we have to ensure that the `created` attribute is not changed after the document is created. We do this by copying the `created` attribute from the most recent version of the document `oldDS` to the new version of the document `DS` before "writing" to back to tuple space.

Embedded Processes: Agents

We now incorporate arbitrary processes into the model that can be used to expand its functionality. We use the term "agent" to describe these processes. We envision three agent types in Lifestreams: stream agents, document agents and personal agents.

Stream agents live on streams and become active when certain events occur (such as the arrival of a new document). Document agents are "attached" to documents via an agent attribute and also become active when particular events occur (such as the user reading the document for the first time). Personal agents typically execute as part of the user's Lifestreams interface and automate user tasks or assist the user. Personal agents are important and we will discuss them in coming chapters, however in this section we will concern ourselves with extending the Lifestreams model to support stream and document agents.

To add agents we build on the work of Borenstein and Rose [BR93]. They developed a model called enabled mailed on top of MIME that allows segments of executable content to be added to standard mail messages (via the multipart and executable

[4] In practice if a document has a creation date and time earlier than `now` we update the date and time to `now`.

[5] In practice we make some allowance for clock drift.

```
proc append(int sid, document DS)
begin
    int docid, docindex, filterindex, valid;

    valid := (extract(sid, docid, "created") >= now)
    if (valid) then
    begin
      in("documenthead", ? docid);
      out("documenthead", docid + 1);
      in("stream", sid, ? docindex, ? subhead);
      out("stream", sid, docindex + 1, subhead);
      out("document", docid, DS);
      out("streamdoc", sid, docindex, docid);
      return docindex;
    end
end
```

Figure 2.13: The (time-aware) definition of append.

content types). Each segment is typed with an event type and, when that event type occurs, the segment is executed. Event types are based on several key states that occur as mail is in transit to a recipient (e.g., the mail arrives at the mail server, the mail is delivered to the mailbox, the mail is read by the recipient).

Similarly we now define several events in the Lifestreams system. While there are an arbitrary number of such events that could be defined, for the purposes of this dissertation we discuss only the following events as they are all that is necessary to demonstrate some interesting system extensions. The `STREAM_APPEND` event occurs any time a document is appended to a stream and all stream agents receive the event along with the document id that was appended. The event `DOCUMENT_READ` occurs whenever a document is read. If the document is being read for the first time then the `DOCUMENT_OPEN` event also occurs.

To support stream agents we add an agent head field to the `"stream"` tuple, an additional primitive, *Add_agent*, a new `"agent"` tuple type and make changes to the *append* primitive. *Add_agent* (Figure 2.15) is a simple primitive that when passed a stream identifier and an agent (code in character string form) adds an `"agent"` tuple to the stream.

The *append* primitive (Figure 2.16) now takes the value of the agent head (bound to `agentnum`) and steps through each agent by **in**ing its `"agent"` tuple (to extract the agent's code) and then applying the agent (through **eval**) to the new document.

```
proc write(int sid, int docid, document DS)
begin
    document oldDS;
    int created;

    rd("streamdoc", sid, docindex, ? docid);
    in("document", docid, ? oldDS);
    created := ds_extract(oldDS, "created");

    ds_replace(DS, "created", created);
    out("document", docid, ? DS);
end
```

Figure 2.14: The (time-aware) definition of write.

```
proc add_agent(int sid, char agent)
begin
    int dochead, subhead, agentid;
    in("stream", sid, ? dochead, ? subhead, ? agentid);
    out("stream", sid, dochead, subhead, agentid + 1);
    out("agent", sid, agentid, agent);
    return agentid;
end
```

Figure 2.15: The definition of add_agent.

To support document agents we need to make additions to the document data structure and to the *read* primitive. To the data structure we add four attributes: `stream_append_agent`, `document_open_agent`, `document_read_agent`, and `lastread`. The first three are attributes for holding the executable content of the agents themselves—one for each event type. The last attribute, `lastread`, holds the access time of the last time the document was read. This allows us to know when the `DOCUMENT_OPEN` event occurs by comparing the value of `lastread` against the value of `created`. Note that no "add_document_agent" procedure is needed because an agent can be added through the *replace* procedure.[6]

Now we only need to make a few additions to the *read* primitive. We will present

[6]The agent can be added via *replace*(`docid`, `document_open_agent`, `agentcode`).

```
proc append(int sid, document DS)
begin
    int docid, docindex, filterindex, valid;
    int agentnum, i;
    char agentcode;

    valid := (extract(sid, docid, "created") >= now)
    if (valid) then
    begin
      in("documenthead", ? docid);
      out("documenthead", docid + 1);
      in("stream", sid, ? docindex, ? subhead, ? agentnum);
      out("stream", sid, docindex + 1, subhead, agentnum);
      out("document", docid, DS);
      out("streamdoc", sid, docindex, docid);
      for(i := 0; i < agentnum; i++)
      begin
         rd("agent", i, ? agentcode);
         eval(agentcode(sid, docindex));
      end
      return docindex;
    end
end
```

Figure 2.16: The definition of *append* with agents.

these changes in the next section when we create a "user interface" to our Lifestreams model.

2.2.5 Constructing a "User Interface"

In this section we show how the stream and document primitives can be used to implement a "user interface" of sorts for the Lifestreams model. For the reader's convenience we reproduce the Lifestreams primitives in Appendix A.

As we mentioned earlier in this chapter, a user's view of Lifestreams can be defined by the operations New, Clone, Transfer, Find, and Summarize. To review, *New* creates a new writable document on the stream. *Clone* takes an existing document and creates a copy on the stream. *Transfer* copies an existing document from one stream to another. *Find* takes a filter and returns a substream based on the filter. *Summarize* takes a substream and a function, and compresses the substream into a summary document

and appends the document to the stream. In addition, implicit in the model is the ability to read and write documents; this functionality already exists in our primitives *read* and *write*, although *read* still needs to be extended to handle document agents. We now present each operation and its definition (Figure 2.17).

New takes one argument, a stream id, `sid`. *New* first creates a new document record, `DS`, and then replaces its `created` attribute with `now` and its `lastread` attribute with `now` -1. Setting `lastread` in this matter ensures that the last access time is less that the creation time (for the purposes of detecting the `DOCUMENT_OPEN` event); although we don't specify the details, *New* also sets any other default attributes that may need to be initialized. Last, *New* appends the document record to the stream `sid` and returns its document index. *Clone* operates similarly. *Clone* takes a stream `sid` and a document index `docindex` and uses *read* to retrieve the document record. *Clone* then alters the document record to make it writable, updates both of its time attributes and then appends it to the stream `sid`. Last, it returns the document index. *Transfer* takes two stream ids, a source stream, `sid`, and a target stream, `sid2`, and a document index, `docindex`. *Transfer* reads the document from the source stream, updates its time attributes, and then appends the document record to the target stream.

Find takes a stream identifier, `sid`, and a substream identifier, `pid`, (the "parent" substream that the find is performed on) and a `query`. *Find* first calls *filter* to create a substream. It then returns the list of documents belonging to that substream by calling *retrieve*.

Summarize takes a stream id, `sid`, and substream, `subid`, along with a summary function `f`. *Summarize* first calls *retrieve* to obtain the substream and then applies `f` to the stream id, the substream, and a document record. The intent is for `f` to "squish" the substream into one document and store it in the document record. Last *Summarize* appends the document record to the stream and returns the newly created document's index. As an example consider a simple summary that counts the number of documents with `document_open` agents on a substream. We define such a summary in Figure 2.19.

Last we define the user interface versions of *write* and *read*. *Write* can be written by using the *write* procedure that we have already defined. In the case of *Read* we still need to add the code to enable document agents.

First *Read* grabs the document and tests to see if the condition of the `DOCUMENT_OPEN` event holds (that is, whether the `created` time is greater than the `lastread` time), if so the `valid` flag is set. We update the `lastread` attribute to disable future `DOCUMENT_OPEN` events. The document is then returned to tuple space. Next if the `valid` flag is set then the `document_open_agent` is extracted and applied to the document. Next, the `document_read_agent` is extracted and applied to the document (since it is applied any time the document is read)[7].

[7] We define an "empty agent" field to be $\lambda sd.d$.

```
proc New(int sid)
begin
    document DS;

    ds_replace(DS, "created", now)
    ds_replace(DS, "lastread", now−1)
    .
    . /* fill in default attributes */
    .
    return append(sid, DS);
end

proc Clone(int sid, int docindex)
begin
    document DS;

    read(sid, docindex, DS);
    ds_replace(DS, "created", now);
    ds_replace(DS, "lastread", now−1);
    ds_replace(DS, "state", true);
    return append(sid, DS);
end

proc Transfer(int sid, int sid2, int docindex)
begin
    document DS;

    read(sid, docindex, DS);
    ds_replace(DS, "created", now);
    ds_replace(DS, "lastread", now−1);
    append(sid2, DS);
end

proc Find(int sid, int pid, char query)
begin
    int subid;
    subid := filter(sid, pid, query);
    return retrieve(sid, subid);
end
```

Figure 2.17: Expression of Lifestreams user interface in terms of the primitives.

```
proc Summarize(int sid, int subid, func f)
begin
     list substream;
     document DS;

     substream := retrieve(sid, subid);
     f(sid, substream, DS);
     return append(sid, DS);
end

proc Write(int sid, int docid, document DS)
begin
     write(sid, docid, DS);
end

proc Read(int sid, int docindex, document DS)
begin
     int docid, valid;

     rd("streamdoc", sid, docindex, ? docid);
     in("document", docid, ? DS);
     if (ds_extract(DS, "created") > ds_extract(DS, "lastread"))
       valid := true;
     ds_replace(DS, "lastread", now);
     out("document", docid, DS);
     if (valid)
       eval(ds_extract(DS, document_open_agent)(sid, docindex));
     eval(ds_extract(DS, document_read_agent)(sid, docindex));
end
```

Figure 2.18: Expression of Lifestreams user interface in terms of the primitives (cont).

```
proc agent_summary(int sid, list substream, document DS)
begin
    int count := 0, doc, docid;
    foreach doc in substream
    begin
        rd ("streamdoc", doc, ?  docid);
        if(extract(sid, docid, "document_open_agent") ≠ "")
         count := count + 1;
    end
    ds_replace (DS, "data",
        "There are " + count + "documents with open agents");
end
```

Figure 2.19: The simple summary.

Extending the User Interface

We now develop several procedures using the document and stream primitives as well as the "user interface" procedures. We start with the *apply* procedure in 2.20, which takes a stream id, `sid`, a substream id, `subid`, and a function `f`, and applies the function successively to each document in the substream. *Apply* does this by first generating the substream (by calling *retrieve*) and then applying `f` to each document in the substream.

```
proc apply(int sid, int subid, func f)
begin
    list substream;
    int docid;

    substream := retrieve(sid, subid);
    foreach docid in substream
       f (sid, docid);
end
```

Figure 2.20: The definition of apply.

Combining the *apply* procedure with *Transfer* we can define *substream_copy* (Figure 2.21), which copies all the documents in one substream to another stream. *Substream_copy* takes three parameters — a source stream identifier, `sid`, a target stream identifier, `sid2`, and a substream identifier, `subid` — and creates a function, `f`, that

takes a stream id, s, and a document, d, as parameters and copies the document via *Transfer* from the source stream to the target stream. Next, *substream_copy* calls *apply* with `sid` (the source stream), the substream `subid`, and the newly created `f`, which results in each document in the substream `subid` being copied to `sid2`.

```
proc substream_copy(int sid, int sid2, int subid)
begin
    func f := λs d.Transfer(s, sid2, d);
    apply(sid, subid, f);
end
```

Figure 2.21: The definition of substream_copy.

Extending Lifestreams

We have proposed that agents can be used to extend the functionality of Lifestreams; in this section we show how this can be done by developing several simple agents in terms of the Lifestream primitives. In Chapter 5 we will show actual agents that extend the functionality of our prototype system.

Return Receipt. A term "return receipt," an idea borrowed from the postal system, is a mechanism that enables the sender of a message to receive a return piece of mail that specifies the time a piece of mail was received by the recipient. In the context of Lifestreams we define "received" to be the time the recipient reads the message at the "user interface" not the time it is appended to his stream. Several commercial electronic mail systems provide this functionality. Here we demonstrate how this behavior is easily added to Lifestreams by means of a document agent. Note that we are using several attributes that we have not previously defined, such as `from`. These attributes are all borrowed from mail systems and their meanings are obvious (e.g., the `from` attribute holds the sender of a message, the `to` attribute holds the recipient, and so on.).

Our document agent, of event type `DOCUMENT_OPEN`, is given in Figure 2.22. When the `DOCUMENT_OPEN` event occurs (as the recipient reads the document) the agent is applied to the document and its stream. The receipt agent first creates a new document called *receipt*, and then extracts the *from* and *to* attributes of the document. The agent then creates a message (a character array) that includes the text "Message read on (date and time) by (recipient)," using the *to* field to obtain the recipient. The data portion of the document *receipt* is then changed to be the character array message. Finally, the agent appends the new document onto the stream of the message's originator (contained in the *from* field).

```
proc receipt_agent(int sid, int docid)
begin
    document message, receipt;
    string body;
    int from, to;

    read(sid, docid, message);
    from := ds_extract(message, from);
    to := ds_extract(message, to);
    body := "Message read at" + now + " by " + to;

    ds_replace(receipt, data, body);
    ds_replace(receipt, created, now);
    ds_replace(receipt, lastread, now - 1);
    ds_replace(receipt, state, false);

    append(from, receipt);
end
```

Figure 2.22: The definition of a receipt agent.

Subscriptions As an example of a stream-based agent we implement a "subscription agent." A subscription agent sits on a stream and copies every new document to another stream. This could be used, for example, to automate an electronic subscription service for an online newsletter or magazine.

We implement subscriptions by reusing the function `f` that we defined in *substream_copy*:

$$\textbf{func}\ \texttt{f} := \lambda s\ d.Transfer(s, \texttt{sid2}, d);$$

As you will recall, `f` takes a stream and a document index a copies the document to `sid2`. Here we define a simple procedure `substream_me` that takes a source stream identifier (the stream we want to subscribe to) and a target stream identifier (where we want to receive the subscription) and installs a stream agent.

When this procedure is invoked, a stream agent is added to the stream `source`; each time an append occurs the agent is evaluated and the new document is passed to the subscriber (the `target` stream).

```
proc subscribe_me(int source, int target)
begin
    func f := λs d.Transfer(s, target, d);
    add_agent(source, f);
end
```

Figure 2.23: The definition of a subscription agent.

2.3 Summary

We have now specified a model for Lifestreams that is capable of supporting a document collection that is chronologically ordered along with dynamic filters that partition the documents into virtual collections. We have presented this model in terms of an "abstract data type" (specified using the Linda coordination language) and implemented a "user interface" on top of the model. In addition, we have provided a means of extending and automating tasks in the model through agents.

In the next chapter we use this model as a starting point for a research prototype. In the process we will take the model from a formal description to a real system that satisfies many of our real-world needs, such as efficiency and compatibility with existing tools.

Chapter 3

The Implementation

Our model in Chapter 2 provides a semantic foundation on which to build a Lifestreams prototype, but it leaves many "systems" issues undefined. Consider for example our definition of substreaming. While it provides a clean semantics that describes the information retrieval and filtering aspects of the substream construct, the definition doesn't address the question of how substreams can be implemented efficiently.

Over the next two chapters we will discuss these issues and describe our efforts designing and implementing a "proof of concept" for the Lifestreams model. Our efforts have proceeded on many fronts, including user interface design, system integration, indexing and retrieval, multi-platform support, agent technologies, universal access, security and performance. The result is a system that is robust enough to see daily use for the last year; nevertheless the system is not a commercial-grade application. We believe it is a good starting point for further research and development (such directions are suggested in Chapter 9).

In this chapter we describe the "behind the scenes" infrastructure that is needed to support Lifestreams. We begin by describing the general architecture and then move on to the various modules in the system. In the next chapter we describe the system from the perspective of the user interface.

3.1 General Architecture

Lifestreams is a network-based system that allows access to a lifestream from any available network-connected client. As such, the Lifestreams architecture is split into client and server parts.

The Lifestreams server is primarily a "storage system" that maintains the streams, substreams and documents of one or more users. This storage system can be conceptually divided into three components: the document collection, a document content index, and the substreams. Our implementation mirrors these components. The server is a single-threaded system that is capable of supporting a number of simultaneous clients

that communicate through a remote procedure call interface (ONR RPC) written on top of TCP/IP. Data is exchanged and stored in a machine-independent format based on the external data representation (XDR) developed by Sun Microsystems [Ste91]. This has allowed us to support servers and clients on several platforms (Solaris, Sun OS, and AIX).

Lifestreams clients include user interfaces, agents and daemons processes that interact with the server. The interfaces provide a way for users to interact with a stream. Agents often interact with a stream to automate tasks. Daemon clients perform monitoring and gateway functions within Lifestreams; for instance a daemon may provide a gateway to and from the Internet mail system. We cover interfaces in detail in the next chapter and we will briefly return to daemons later in this chapter.

3.2 Server Infrastructure

We now describe the server's three storage systems: the basic "document collection" module that maintains and provides access to the documents, the indexing system that supports efficient content-based search, and the substreams mechanism.

3.2.1 Document Collection Subsystem

The document collection is the simplest subsystem within the server; it maintains an addressable set of documents and implements document access functions. Each document's attributes are grouped into a record with one field per attribute (recall that documents are represented by attribute/value pairs). These records are stored in a dynamic array that we call the "skeleton" (Figure 3.1). Each field stores the value of its attribute (e.g., the `state` field records a boolean value) with one exception. The `data` attribute, which represents the actual content of the document, stores a "pointer" to its value rather than the value itself. Given that the document content may be quite large (possibly megabytes), swapping the skeleton in and out of main memory is prohibitive if it contains the contents of all documents. Instead we store the document content on disk and its file path in the `data` attribute and swap the contents in only when needed. The skeleton itself is also swapped in and out of core according to the server's needs. We currently swap the entire stream *in toto*; in order to support large lifestreams (hundreds of thousands of documents) the server will need to employ a paging scheme.

The document subsystem provides three points of access to the documents: `dc_read`, `dc_write` and `dc_append`. These functions provide the functionality described in Chapter 2: `dc_read` retrieves a specific document by its index in the skeleton, `dc_write` replaces a document in the skeleton (assuming it is writable), and `dc_append` adds a new document to the skeleton by dynamically extending the array by one and adding the document to the end slot. We will see how the chronological stream and substreams interact with the skeleton shortly.

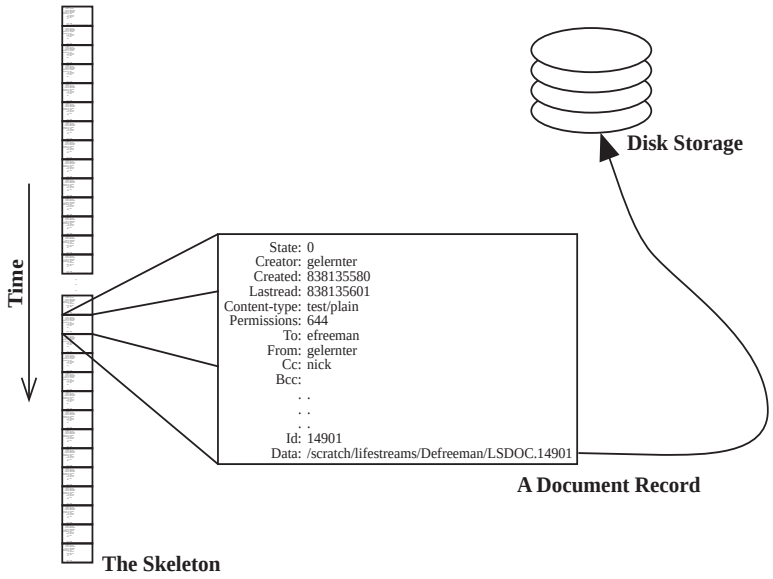


Figure 3.1: The Document Storage Subsystem.

Each of these functions requires a security check. Our current security model is loosely based on the UNIX security mechanism. Each document record contains a field called `permissions` that is represented by a bitfield and describes who may read and write the document. As in UNIX, the granularity of control is owner, group, and world. Each Lifestreams request is accompanied by a "credential" that describes the requester; this is compared against the values in the bitfield before access is allowed.

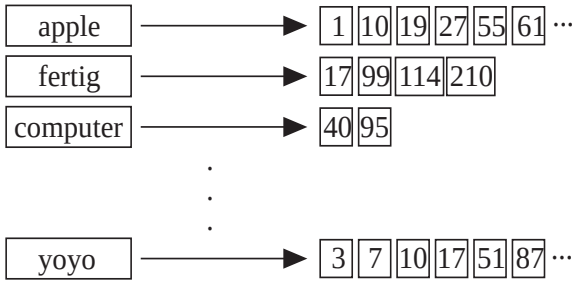
3.2.2 Indexing Subsystem

Efficient and accurate computation of substreams is crucial to the success of Lifestreams. We have already argued that naming and static directories are sub-optimal means of organizing information. We can derive a more effective means of retrieval from the information already inherent and readily available in a document collection (as shown in [GJSO91]), namely the document's content.

How do we achieve content-based retrieval with reasonable performance? Brute force sequential search is not sufficient given the time it requires. For example, a one-keyword search over text documents on a Sun Sparc 10 takes on the order of one second per megabyte. Given a lifestream of forty megabytes (arguably a small stream) a `find` operation would keep the user waiting forty seconds. This is for one keyword; we often want to search for more complex queries (e.g., "david and lifestreams and (not scott)"). Such a query could take minutes using brute force methods.

Information retrieval strategies for large static document collections have been under development for the last decade or so. In general these systems have traded space

and storage time for faster performance at retrieval time. Most modern systems use some form of inverse index [FE92]. Inverse indices or inverse files map from search terms to documents. For example, in the inverse index depicted below, each word maps to a set of document identifiers.



Keyword lookups are performed by retrieving the list of documents that match the keyword via the inverted file. By maintaining an inverted file of a document collection we can perform content-based retrieval in time proportional to the number of the keywords in the query[1]. Finding the keyword entry in the inverted file is usually accomplished through a hash table or B-tree [FE92]. Complex queries, such as boolean expressions, are accomplished by taking the intersection, union and difference between document lists retrieved from the inverted file.

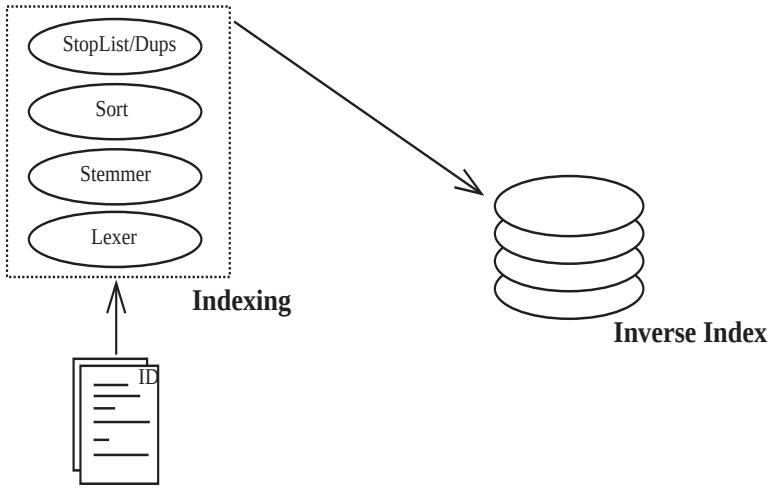


Figure 3.2: The Indexing Subsystem.

The Lifestreams' server maintains an inverted file for each lifestream and indexes a document through the `in_index` procedure. Figure 3.2 shows a conceptual picture of the indexing system. New documents are parsed and their contents are added to

[1] With the implementation techniques discussed later in this section, query computation has an expected running time of $O(N)$ per boolean operation, where N is the number of documents in the collection.

the inverted file (we will describe the process shortly). The current prototype indexes English ASCII text files. For non-text documents we still index "header" information and user supplied attributes. Other work in the field of information retrieval is exploring multi-lingual and image-based indexing methods [Flu, BFJ+95]. Our indexing methods are currently of the most basic sort, but we believe that if Lifestreams is useful, even using limited techniques, it can only prove more useful with more powerful indexing technologies. Moreover, given that we have designed Lifestreams as three "modular" subsystems, more advanced indexing technologies can easily be added. We now describe our indexing procedure in detail.

Lexing, Stopwords and Stemming

Parsing a text file for inclusion in the inverted file is a multi-step process. To create the inverse index of a text document we first break the text into tokens. In the Lifestreams server we do this by treating all punctuation and whitespace characters as delimiters. Any delimited piece of text is treated as a token.

We then convert tokens into canonical form by applying the Porter algorithm [Por80] for suffix stripping. This algorithm is one of a class of algorithms for stripping the suffixes (and possibly prefixes) from a word, leaving a stem. For instance, stemming the words "runner," "running," and "run" all result in a stem of "run." There are two benefits to stemming words: first, the overall size of the index can be decreased substantially (on the order of 30% [FE92]). Second, retrieval can be more effective as documents that contain words with common stems will satisfy the same search query. The disadvantages are that retrieval is not as precise (although, in some respects this is beneficial, as noted above) and indexing in general is not easily amenable to more flexible "wildcard" searching — such as using regular expressions as keywords or allowing for misspellings. Ongoing work (in the information retrieval community) is however making progress in these areas by combining fast (Boyer-Moore) searching with course-grained indexing [WM94, MW93].

We then sort the list of tokens and remove any duplicate tokens. Duplicates may occur in the document or may result from the stemming process (where words with different suffixes may result in the same word after stemming).

The next step in the indexing process is to remove noise. Text contains a large proportion of words that occur too frequently to be of use for retrieval purposes. Because the words appear in a large percentage of documents, their use as search terms is non-discriminating. Removing these words before indexing significantly reduces the total amount of words that need to be indexed and the overall space requirements for the disk index [FE92]. Examples of stopwords are:

```
a          although    become     co         enough     four
about      always      been       could      etc        from
above      amongst     before     couldn't   even       further
```

```
according   an          beforehand  d         ever       g
across      another     begin       did       every      h
actually    any         behind      didn't    everyone   had
adj         anyone      below       do        everything hasn't
after       anything    beside      does      everywhere have
afterwards  anywhere    between     doesn't   except     haven't
again       are         beyond      down      f          he
against     aren't      billion     during    few        hence
all         around      both        e         fifty      her
almost      as          but         each      first      here
alone       at          by          eight     five       hereafter
along       b           c           either    for        hereby
already     be          can         else      formerly   herein
also        became      cannot      elsewhere forty      herself
at          because     caption     ending    found      him      ...
```

These words are mainly common verbs, pronouns and prepositions. Lifestreams maintains a system-defined "negative dictionary" or "stoplist" of these words (our stoplist was taken from the words in the WAIS stoplist [Kah91]) and in the lexing process compares each token against the stopwords. This stoplist resides in a hashtable (i.e., each stopword is a hashtable key) and a lookup is performed on each token. If the token is found in the hashtable then it is discarded. Otherwise it is added to the inverse index (as described below). One system-wide stoplist is currently maintained by the server and so the same negative dictionary is applied to all users. Future systems should be able to employ custom dictionaries for each user.

Last, our indexing system allows keywords to be *tagged*. For instance, a keyword appearing in the title of a document would be tagged as a "title" keyword. This tag can then be used in search queries to identify documents with a specific keyword in their title. Lifestreams automatically performs tagging based on several attributes, such as the content type (image, text, audio, etc.) and mail-related attributes (to, from, subject).

Processing Documents

To summarize; to index a document *id* we:

1. Lex the document into a list $\mathcal{T}$ of tokens.
2. Stem all $t \in \mathcal{T}$.
3. Sort $\mathcal{T}$ and remove any duplicates.
4. Given a stoplist $\mathcal{S}$, we let $\mathcal{T} = \mathcal{T} - \mathcal{S}$.
5. Add *id* to the document list of each $t \in \mathcal{T}$ in the inverted index.

Computing Search Queries

When a `find` operation is initiated by a Lifestreams user, the server performs a search. We now briefly describe the general task of computing a list of documents from a search query. Lifestreams currently supports general keyword searches and searches via boolean queries (keywords composed with `and`, `or` and `not`). There are other models for providing a search capability in Lifestreams, such as database query languages (too complex and cumbersome for the average user) and Shneiderman's dynamic queries [Shn94]. Shneiderman's work suggests an interesting method of searching over a stream via slider bars, but the idea still needs work to scale beyond small databases (currently on the order of 1000 records or less).

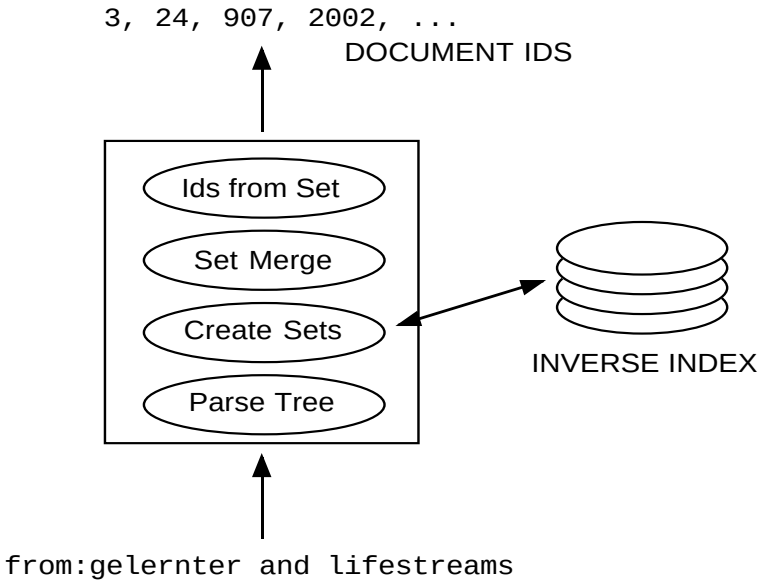


Figure 3.3: In_search: computing a substream.

Lifestreams' index subsystem provides computation of search queries through the function `in_search`, which takes a boolean search query and a stream and returns a set of documents that satisfy the query (Figure 3.3). The `in_search` function is implemented as follows: First, each query is parsed into a parse tree. For each leaf node (keyword) a set data structure is created[2]. Sets are implemented with hashtables (a common technique in information retrieval systems). For each set a hashtable is created and is "keyed" with every document identifier contained in the inverse file for the corresponding keyword. By "keyed" we mean that each document identifier is added as a key to the hashtable. Hashtables as an implementation strategy have the advantage that they are more efficient in their use of space than other approaches

[2]This could also be accomplished (perhaps more efficiently) by searching through the document directory.

(such as bit-vectors), do not limit the maximum size of the document collection and have good expected running time characteristics (see [FE92]).

Once we have created a set for each keyword in the search query, the operators are applied depth-first: `and` is accomplished by taking the intersection of two sets , `or` by the union, and `not` by subtracting the set from a set that contains all documents in the collection. The result is one set which contains the documents that satisfy the search criteria[3]. A similar technique is used when filtering documents into existing substreams (discussed below).

We will present data on search times in Chapter 6.

3.2.3 Substreams Storage Subsystem

The substream subsystem manages the substreams of a stream, providing support for the creation of virtual document collections and continual information filtering via persistent substreams. Here we follow the model in Chapter 2 by treating the main stream (all documents in the lifestream) as just another substream. Note that the skeleton is not a good candidate for the main chronologically-ordered stream because it is ordered by each documents "real" creation time, not its time within the Lifestreams system. In Lifestreams we can reset the time of the system to the future and create documents there. The skeleton orders documents by physical creation time; we need a data structure for the chronological stream that orders documents by virtual time.

For this data structure we use a dynamic array of indices that point back to the document skeleton. The substream data structure holds this array, a copy of the search query and a parent substream identifier (we will see how the search query and parent identifier are used in the next section). In Figure 3.4 we show three substreams: the main substream (substream 0) has a parent identifier of 0, a search query of `"*"` (recall the F_0 substream from Chapter 2) and an array that contains a document index for every document in the collection. The other two substreams are represented in a similar manner, although they may contain only a subset of the documents in the collection. Note that substream 1 has a parent identifier of 0 (the main stream), while substream 2 has a parent identifier of 1 (as it was created incrementally from substream 1). All the document indices in each substream are sorted by the "virtual" creation time of each document in the skeleton.

A substream is created by calling `in_search` with the search query. The (possibly empty) list of documents returned from `in_search` is then installed (sorted by creation time) as a new substream along with the search query and parent identifier.

[3]Later in this chapter we will see that `in_search`, in order to achieve incremental substreaming, also takes a parent substream id and `AND`s the results with the parent substream before returning the set of documents.

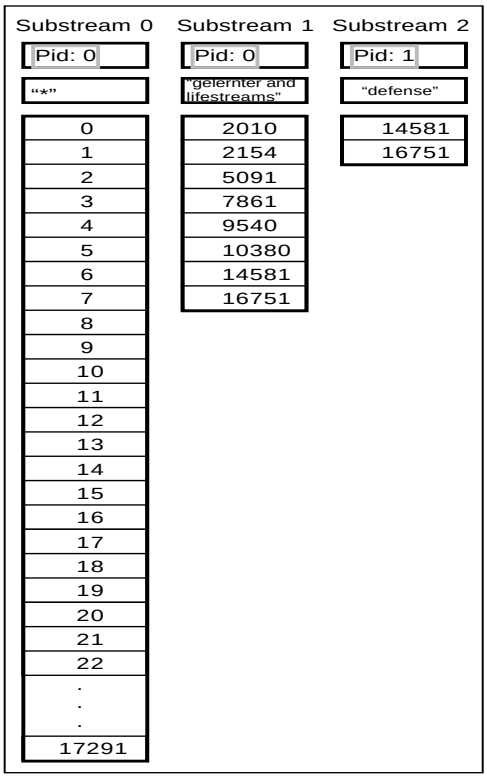


Figure 3.4: The Substream Data Structure.

Filtering New Documents

When a new document arrives or is created, it is added to the document collection via `dc_append`. Next, `in_index` is called and the document is properly indexed. Last `ss_process_substreams` is called and, for each substream, the corresponding search query is applied to the new document. If the document passes the search query, then it is added to the end of the substream.

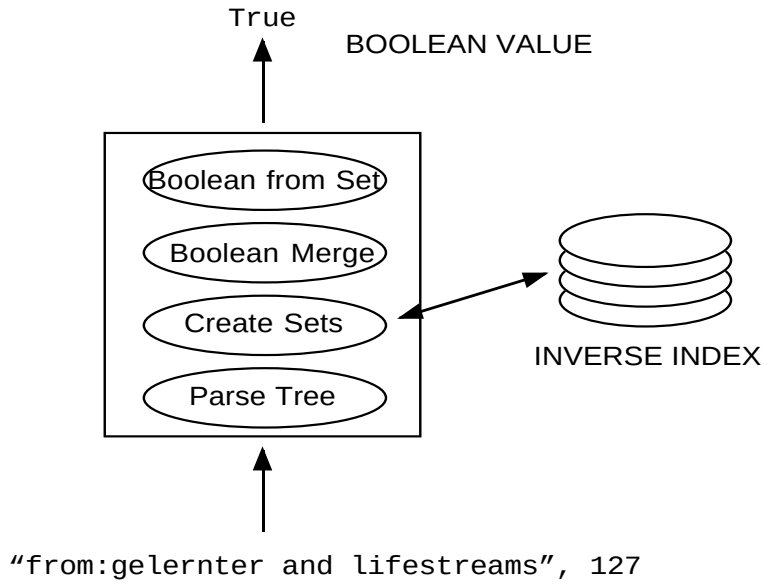


Figure 3.5: In_match: computing a documents membership in a substream.

To apply a search query to a document (rather than a document collection) we call `in_match` (see Figure 3.5). This function operates analogously to `in_search`: First, each query is parsed into a parse tree. For each leaf node (keyword) we retrieve the document list. If our new document exists in that document list, we replace the leaf node with a boolean `true`, otherwise with a `false`. The boolean operations on the interior nodes are then processed, resulting in a boolean value. If the resulting value is `true`, the new document is added to the substream.

3.2.4 Putting it All Together

We now present the server as a whole by describing how the subsystems work together. We do so by supplying the implementation details for the stream primitives (*read*, *write*, *append*, *retrieve* and *filter*) that we developed in Chapter 2. Each of these procedures are implemented as remote procedure calls that can be invoked from a client.

```
struct document *Read(int sid, int docid) {
    return dc_read(sid, docid);
}

void Write(int sid, int docid, struct document *doc) {
    dc_write(sid, docid, doc);
}
```

Figure 3.6: Definition of the Server Read and Write Routines.

Read/Write

Read (Figure 3.6) is implemented directly through the document storage subsystem. **Read** takes a stream identifier and a document identifier and calls `dc_read`, returning its result. Similarly, **Write** (also Figure 3.6) takes a stream identifier, a document identifier and a document record and calls `dc_write`.

Append

Our definition of append in Chapter 2 adds a document to a stream.[4] In our prototype, **append** (Figure 3.7) takes a stream identifier and a document record and calls `dc_append`. Recall that `dc_append` extends the skeleton (the document collection, not the main stream) by one and adds the document record. The data field of the document is swapped out to disk in the process (to reduce core memory usage). **Append** must also take care of substream and inverse index housekeeping: **append** calls `in_index`, which adds the document contents to the inverse index. **Append** then iterates over the all existing substreams and calls `in_match` for each one. For the calls of `in_match` that return `true`, the document `id` is added to the appropriate substream. Note that the one exception is substream 0 (the main stream), which always passes the conditional. Last, **append** returns the document `id`.

Filter

Filter takes a stream identifier, a parent substream identifier and a search query and creates a new substream. **Filter** (Figure 3.8) first creates a structure and an array to hold the new substream. The structure holds the array, the search query and the parent substream identifier. **Filter** then calls `in_search` with the search query and adds all document returned to the substream's array. Last, it returns a new identifier for the substream. If the reader returns to the definition of **append** above it should become apparent how the substream is built over time as new documents arrive.

[4] It also supports stream agents, which our research prototype does not directly support.

```
int Append(int sid, struct document *doc) {
    int docid, len, i;

    docid = dc_append(sid, doc);
    in_index(docid, doc);

    for(i=0;i< skeleton[sid].subnums; i++) {
        if (i == 0 || in_match(sid, docid, i)) {
            /* dynamically extends by one document */
            len = skeleton[sid].substream[0].len++;
            skeleton[sid].substream[i].docs =
                realloc(skeleton[sid].substream[i].docs,
                        len * sizeof(int));
            skeleton[sid].substream[i].docs[len-1] = docid;
        }
    }

    return docid;
}
```

Figure 3.7: Definition of the Server Append Routine.

```
int Filter(int sid, int pid, char *sq) {
    int subid, len, *docs;

    len = in_search(sid, pid, sq, docs);

    subid = skeleton[sid].subnums++;
    skeleton[sid].substream =
            realloc(skeleton[sid].substream,
                    subid*sizeof(struct substream));
    skeleton[sid].substream[subid].len = len;
    skeleton[sid].substream[subid-1].sq = strdup(sq);
    skeleton[sid].substream[subid-1].pid = pid;
    skeleton[sid].substream[subid-1].docs =
            malloc(sizeof(int) * len);

    for(i=0;i<len;i++) {
        skeleton[sid].substream[subid].docs[i] = docs[i];
    }
    return subid;
}
```

Figure 3.8: Definition of the Server Filter Routine.

```
int *Retrieve(int sid, int subid) {
    int len, i;
    int *docs;

    len = skeleton[sid].substream[subid].len;
    docs = (int *) malloc(len * sizeof(int));
    for(i=0;i<len;i++) {
        docs[i] = skeleton[sid].substream[subid].docs[i];
    }
    return docs;
}
```

Figure 3.9: Definition of the Server Retrieve Routine.

Retrieve

Retrieve (Figure 3.8) is a simple procedure: it takes a stream identifier and a substream `id` (returned from the call to **filter**) and returns the document identifiers pointed to by the array of the corresponding substream.

We spent a good deal of time in Chapter 2 describing how substreams can be incrementally created from other substreams. The reader may ask how this works in our research prototype. It is implemented as follows: the substream data structure maintains a list of substreams and their "parent" substreams. When `in_search` and `in_match` are called, they `AND` the result with the parent substream. The result is incremental substreaming.

3.2.5 Reality Check

The efficiency of our indexing code is important to the success of our prototype and we would like to be in the "ballpark" when compared to existing commercial indexing systems. Here we provide a comparison with the WAIS indexing system [Kah91] (now called Isearch) which has been widely used for commercial and academic information retrieval. Although it is a more advanced indexing system, WAIS is similar to our system.

For our tests we indexed a 21.8 megabyte collection of newsgroup articles from the `soc` newsgroup hierarchy using WAIS and the Lifestreams server. The results (gathered on an unloaded Sun Sparc 10) are provided in Table 3.1. We see that while Lifestreams was roughly 25% faster indexing the collection it used about 60% more space. This is to be expected given some of the differences between the two systems. With respect to indexing time, WAIS collects more information while indexing (word adjacency, number of occurrences per document, etc.), which increases the time to parse and index the

	Lifestreams	WAIS
Indexing Time (seconds)	1562	2025
Indexing Speed (MB/min)	.836	.645
Index Size	17.1MB	10MB
% of Collection	78.5%	46.3%

Table 3.1: Comparison of WAIS and Lifestreams indexing.

documents. With respect to index size, WAIS and Lifestreams operate in two different manners. WAIS operates in a batch-like mode and precomputes the entire index to use space efficiently, while Lifestreams incrementally builds up its index as documents are added and must set aside extra space for future use. In any case, Lifestreams' indexing time is slightly faster than WAIS (although the index times would be closer if we added some of WAIS's more advanced parsing) and its use of space (about 80% of the collection) is well within the norm for information retrieval systems (which range from 50% to 300% of the information collection [FE92]).

3.3 Client Infrastructure

Our client architecture relies on a specific document model, method of transferring documents and "embedded computation" technology. In this section we first introduce our document model. We then describe how the Lifestreams client implements the transfer operation. Next we present our current agent model and describe how embedded computations are accomplished. Last, we describe the "plug-in" architecture of the summarize operation (also an embedded computation).

3.3.1 Our Document Model

Early on, we implemented a Lifestreams system that relied on the Andrew Toolkit's built-in editor and multimedia document standard (ATK) [Bor]. While the ATK provided an interesting initial environment in which to explore Lifestreams, it made user migration to the system difficult. Users relied on specific document types (text, PostScript, DVI, GIF, etc.) and they found it inconvenient to create these documents using the Andrew Toolkit's builtin editor rather than with conventional applications.

Our current system is based on the "hypertool model" [Ous94] and MIME types [BF92]. The hypertool model is a method for building complex systems out of smaller reusable applications. We use existing editor and viewer tools in concert with the Lifestreams system so that users can continue to use the applications they are accustomed to. MIME is a document typing standard that grew out of work on Multipurpose

Internet Mail Extensions (MIME) [BF92]. The MIME standard describes extensions to conventional mail systems that allow message content other than 7-bit ASCII to be transmitted through mail messages. This is accomplished by adding a content-type header field to mail messages along with an encoding standard that represents arbitrary binary files as 7-bit ASCII. In addition, MIME provides conventions for adding new types. A MIME type is specified as a content type and subtype; some of the more common MIME-types are plain text (text/plain), GIF images (image/gif), and Quicktime video (video/quicktime). We make use of MIME to support multiple document types within Lifestreams by adding a `content-type` attribute to each document. The mapping from types to applications is user defined. We will describe this mapping and other details of the hypertools model in the next chapter.

3.3.2 Communication

The transfer operation provides a means of sending a document from one stream to another. This includes the capability of transferring non-ASCII content (images, sounds, movies), attached document agents, and "future" documents. To be useful as a general communications tool, transfer must also be able to interoperate with the non-Lifestreams world and be capable of sending documents as conventional electronic mail messages.

The implementation of transfer has evolved over the course of system development. Initially, transfer was implemented by calling **append** on the target stream. There are three problems with this approach: first, it doesn't satisfy our interoperability requirements (because it doesn't work with conventional electronic mail); second, the system did not implement a true "store and forward" communications system[5] so the client expected the target server to always be available (something we cannot guarantee with the current system and something few software systems and networks can guarantee). The third problem was technological and sociological: users were forced to maintain two name spaces, one for Lifestreams and one for conventional mail (for example the author's email address is freeman-eric@cs.yale.edu while his Lifestreams' address is efreeman@pythagoras.cs.yale.edu); that is, the user had to remember which addresses designate conventional mail addresses and which addresses designate Lifestreams addresses and specify transfer operations accordingly. This causes a related problem: when the user of a conventional mail system replies to all the senders of a message containing a Lifestreams address, his mailer won't have the necessary protocol to talk directly with a Lifestreams address. In a future Lifestreams utopia such problems might not arise, but in accommodating our local users it had to be solved.

Our second (and final) implementation uses conventional mail protocols to transport both Lifestreams documents and Internet-bound mail. This approach solves our interoperability requirements, unifies the email/Lifestreams namespace and provides a

[5] It has been argued that no mail system is usable in the real world without a store and forward capability.

robust store and forward communications infrastructure. Let us see how it is implemented.

Communications Implementation

When a transfer operation is executed the client first converts the Lifestreams document (the attribute/value record) into a form that can be shipped within the SMTP mail protocol. This is accomplished by creating a serialization of the document (complete with its `data` field) into XDR form. Once the document is serialized we can convert it into a base-64 representation that contains only ASCII characters and thus can be shipped over conventional SMTP channels. The base-64 representation entails a 33% increase in the space requirements for the document.

The client then creates a mail message buffer that is typed as a `mixed` and `multipart` MIME message. This type tells a MIME reader that the message contains multiple parts that are of different types. If possible, the client first provides a readable form of the message for non-Lifestreams readers.[6] The client then adds the base-64 representation and types it `application/x-lifestreams` and the file is handed to `sendmail` for delivery. Figure 3.10 displays one such message. This message contains a `text/plain` MIME type, the contents of which can be seen in the first part of the message. This document was created "in the future" and so needed to be encoded so that it is placed in the future of the recipient's stream. The base-64 encoded version is seen in the second part of the MIME message.

We also need one additional piece of software: a standalone piece of code that intercepts a user's Internet mail, converts it to a Lifestreams document and forwards it to his lifestream. The conversion program works as follows: when the program receives a mail message, if it contains a "applications/x-lifestreams" part, then that part is unpacked and forwarded to the appropriate Lifestreams server through **append**. In this way we still rely on **append** to place the document on the target lifestream, however we "piggy back" on top of the Internet mail system to actually deliver the document. If the document does not contain a "applications/x-lifestreams" part then we add it to the stream as a text document (parsing it first to extract attributes). To the Lifestreams user the entire process is invisible.

The advantage of this approach is that we can use a long-tested, robust delivery system for transferring Lifestream documents. In addition, communication over these channels is transparent (to the user) and the user only needs to manage one namespace. The disadvantage is that our system is less integrated from the standpoint of software design (the transfer capability is no longer implemented purely as a Lifestreams RPC calls) and harder to maintain from the perspective of system administration (we have

[6] In the case where the message is purely text and there is no Lifestreams specific content (e.g., agents, the document is a future document, etc.) the encoding process is skipped and the mail is sent out as a conventional ASCII mail message.

to manage two additional pieces of software: sendmail and the software that unpacks Lifestreams mail message).

3.4 Embedded Computation

Lifestreams uses "embedded computation" as an implementation strategy for several parts of the system: document agents, personal agents and our summarize architecture. While embedded computations could also be used for stream agents, our implementation doesn't yet support them (although we will suggest how it could later in this section).

Embedded computations are (typically) small pieces of code that are loaded and executed within a running program. In the case of a document agent the code is extracted from the appropriate document attribute and executed within the user interface. A personal agent is loaded into the user interface at "startup" time and executed when the user indicates he wants the agent to run (we will see specifically how this is done in the next chapter). Finally, the summarize functionality, like the personal agents, is loaded into the client at startup time.

In this section we first introduce our choice of a language for writing agents and summarizers (from here on we will call this the "agent" language). We begin by describing our requirements and the reasons for choosing an existing scripting language. We then describe how embedded processes are supported in the Lifestreams client.

3.4.1 The Agent Language

Agents are generally described through a programming language, or through a user interface that translates graphical descriptions into an underlying language. In Chapter 2 we used an Algol variant to describe agents. While in theory any programming language would do, there are a few real world requirements that we would like our language to meet given that agents are executed in an embedded environment.

1. **Heterogeneity:** The agent language should be portable and efficiently executable on virtually computer system. This requirement is necessary so that Lifestreams clients on different platforms can send and receive agents.

2. **Extensibility:** The agent language should be extensible so that we can add Lifestreams functions to the language.

3. **Dynamic binding:** The agent language should be able to dynamically bind Lifestream functions so that an agent can be loaded in place and executed using local versions of Lifestreams functions. This reduces the size of the agent itself (it does not have to carry around Lifestreams libraries with it) and allows the end user to use his own implementations of the Lifestreams functions.

```
From freeman-elisabeth Mon Mar 25 12:26 EST 1996
Date: Mon, 25 Mar 1996 12:26:02 -0500 (EST)
From: freeman-elisabeth
Message-Id: <199603251726.MAA05490@TEDDY.SYSTEMSY.CS.YALE.EDU>
Subject: (Fwd) hotel
To: freeman-eric
X-Mailer: Lifestreams 1.1a1 (Yale)
Content-Type: multipart/mixed; boundary="-"
Content-Length: 1002

This message originated from a Lifestreams client, which has
taken care to provide a text readable version.  Also included in
this message is a MIME-encoded version of the Lifestreams document.  To
properly view it use your favorite Lifestreams' client.
---

Remember to check on hotels near Seattle airport for CHI trip

---
Content-Type: application/x-lifestreams

AAAAAG1iZXIAAAJYMWKScAAAAAAxYpJvAAAACnRleHQvcGxhaW4AAAAAAAEAAAAAAAAAPlJl
bWVtYmVyIHRvIGNoZWNrIG9uIGhvdGVscyBuZWFyIFNlYXRObGUgYWlycG9ydCBmb3IgQOhJ
IHRyaXAKAAAAAAAZZnJlZW1hbi1lcmljQGNzLnlhbGUuZWR1IAAAAAAAAAAhiZnJlZW1hbgAA
AAAAAAAAAAACyhGd2QpIGhvdGVsAAAAAAAAAAA/AAAAUgAAAGUAAABtAAAAZQAAAGOAAABi
AAAAZQAAAHIAAAAgAAAAdAAAAG8AAAAgAAAAYwAAAGgAAABlAAAAYwAAAGsAAAAgAAAAbwAA
AG4AAAAgAAAAaAAAAG8AAABOAAAAZQAAAGwAAABzAAAAIAAAAG4AAABlAAAAYQAAAHIAAAAg
AAAAUwAAAGUAAABhAAAAdAAAAHQAAABsAAAAZQAAACAAAABhAAAAaQAAAHIAAABwAAAAbwAA
AHIAAABOAAAAIAAAAGYAAABvAAAAcgAAACAAAABDAAAASAAAAEkAAAAgAAAAdAAAAHIAAABp
AAAAcAAAAAoAAAA

-----
```

Figure 3.10: An example MIME-encode Lifestreams document.

4. **Safety:** Agents have been called "good viruses" [Way95]. As with any virus, there is also potential for harm, especially when the creator of an agent is malevolent. We will not present a solution to safety *per se* in this dissertation, as there are already a number of good solutions to this problem.

This list is not exhaustive; there are other issues to consider — methods of agent verification, performance, the development environment, and the type of native code used in the underlying system. We have tried to choose the most important issues to consider, at least within the scope of our prototype system.

A few years ago one might have created a special purpose agent language from scratch, but today there are a number of existing and emerging languages that are appropriate as agent languages and that meet our requirements: Sun's Java, variants of Scheme, Telescript, and Tcl-Tk [Ous94] are some of the prominent examples. Each has its own advantages and disadvantages; we have chosen Tcl-Tk as an initial agent language for the following reasons:

1. Tcl is an ASCII-based, interpreted language that runs on most platforms.
2. Tcl is easily extended via the C programming language (used for the core Lifestreams communication routines).
3. Tcl can be loaded in place and dynamically bound.
4. Our user interface is coded in Tcl-Tk.
5. Tk provides a easy method of implementing user interfaces within a Tcl script.
6. Tcl provides a facility for embedding Tcl interpreters in C code.
7. Work has already been done on a "safe" version of Tcl for agents [Bor93b].
8. Tcl/Tk is freely available.

While Tcl meets our needs for the purposes of this prototype, there is nothing to prevent an alternative agents language from being used in the future.

3.4.2 Implementing Embedded Processes

From an architecture perspective, embedding processes works as follows: referring to Figure 3.11, the client interface relies on an underlying Lifestreams library that contains a number of communication primitives as well as primitives for creating, altering, and viewing documents. A Tcl-Tk interpreter, which runs in the client interface's address space, is used to evaluate all agent scripts and, through the Lifestreams library, has access to the communication and document specific primitives. Note that our client is itself written in Tcl, so the agent can be evaluated directly within the user interface.

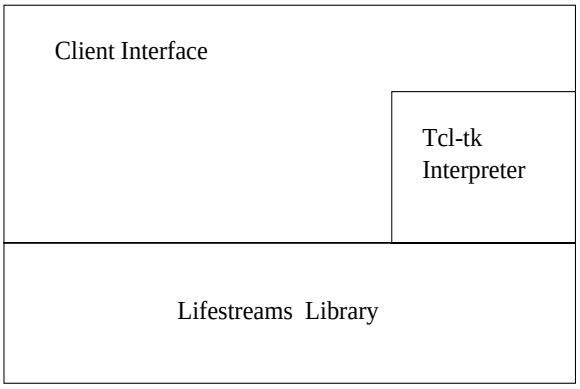


Figure 3.11: Embedded Process Architecture.

As we have mentioned, each document and personal agent is stored in a data structure (e.g., in the attribute of a document, or attached to the user interface). Document agents are evaluated when a specific event occurs, (Chapter 2). Personal agents are attached to the user interface and evaluated whenever the user selects the agent. These agents run within the client's address space.

Document agents can be attached to any document by supplying one or more scripts in the appropriate attribute. When a document is first opened (when the value of the `lastread` attribute is greater than the value of the `created` attribute), the `document_open_agent` attribute is checked for scripts. If they exist they are executed in an arbitrary sequential order. After the open scripts have been executed, or if the document has been opened previously, the `document_read_agent` attribute is checked for scripts. If they exist they are executed in the same manner as the open scripts.

Personal agents are attached to the user interface when the Lifestreams client starts up. Currently a list of these agents is kept in a startup file (although it could be moved to the stream itself). For each agent a menu item is created under the `Personal Agents` menu (see Chapter 4) and its code is loaded into the client for later execution.

Although we don't currently support server agents, it is useful to consider how they will be supported in future versions. The server is written entirely in C code. To run an agent it would need to create a Tcl interpreter and run the agent within the interpreter. This has the added advantage of encapsulation; that is, if the agent halts in some violent way, the server would be unaffected. Future versions of Lifestreams will include multi-threaded servers that could run agents in lightweight processes.

3.5 The "Summarize" Architecture

Recall from Chapter 2 that the "summarize" operation allows users to distill information from a large number of documents by compressing a substream into an overview document. The content of the overview depends on the type of documents in the sub-

stream. While we will provide some examples of preliminary summarizers (Chapter 5) the techniques used to distill this information into something meaningful are beyond the scope of this dissertation and belong to such fields as natural language processing [Jon] and data mining [HS94]. In this dissertation we provide a storage system and general purpose "summarize" architecture that encourages the development of summarizers.

Each summarizer is a software module that is added to the client through a "plug-in" architecture. Like agents, they are embedded computations. Unlike agents (specifically personal agents) they provide a specific functionality: a boolean tester function and a summarize function that returns a document to be added to the Lifestream. Several summarizers come pre-installed in our software client, while new summarizers can be added via the plugin architecture by advanced users. Each summarizer is written in Tcl/Tk, loaded into the client at startup, and follows the conventions of the following plugin API: Each summarizer consist of two procedures, `test` and `summarize`. The `test` procedure takes a substream as an argument and returns a boolean value indicating whether or not the summarizer is appropriate for the substream. The API does not specify what the contents of the `test` procedure must be, but it will usually contain a heuristic for deciding if the summarizer is appropriate for the substream. The `test` procedure should have a short running time (less than a second[7]), although no limit is enforced in the current implementation. The `summarize` procedure takes a substream as an argument and returns a (possibly empty) document.

Both procedures are installed into the client by `InstallSubstreamSummary`. This call takes four arguments: a name and description of the summarizer, a `test` procedure and a `summarize` procedure.

Given that one or more summarizer may be appropriate for a given substream, how does the Lifestreams client decide which one is appropriate? We use a "delegation" procedure: when a user specifies that he wants to summarize a substream (see Chapter 4), the client iterates over the summarizers and calls each `test` procedure; for each `test` that returns true, a pointer to its `summarize` procedure is added to an array[8]. After completing this iteration, if the array is empty a default system summarizer is applied to the substream. If the size of the array is one (meaning that exactly one summarizer is appropriate), that summarizer is applied. If the size of the array is greater than one, the user is asked to choose the summarizer he wants (via a list of descriptions). If the document returned from the summarizer is non-empty then it is added to the stream and displayed for the user.

[7] Future versions may ease this restriction by allowing summarizers to precompute their test in the background.

[8] This information may be cached for future summarize operations on unmodified substreams.

3.6 Summary

Lifestreams is up and running on our local computing environment — a collection of SunOS, Solaris, and AIX workstations. Our initial implementation efforts have focused on providing a "proof of concept" of the Lifestreams model. The server is limited in that it is single-threaded (and thus has a single access point). While the server has been reasonable for a small number of users, clearly a multi-server and multi-threaded approach would be more scalable. To overcome any current scalability problems we distribute our user's streams over several servers, placing two to three streams on each server. In this configuration there have been no major performance complaints from users (see Chapter 6); over the last twelve months of the project the system has been responsive and robust, being responsible for only three down intervals.

There is still work to be done to make Lifestreams scalable; results from the information retrieval and database communities are encouraging. Lifestreams incorporates ideas from both disciplines (like database management systems, Lifestreams manages a database of records (documents), like information retrieval systems, Lifestreams allows access to those records via content-based retrieval). We've seen no real performance problems with respect to retrieval and, given the very large indices that are being used in systems like Nexis/Lexis and Internet search engines, we believe our retrieval scheme should scale to large document collections. These system typically contain tens of millions of index documents and are able to compute user queries within seconds.

Both client and server store the records of the entire document collection when a user views his Lifestream; we need to borrow database technology for large collections. There are "human-computer interaction" problems to solve too. Since no user can look at 10,000 documents at once and discern any usable information,[9] it doesn't make sense to give users an entire document collection at once. A more reasonable approach would be to use "cursors" to allow the user to view segments of the document collection and to load in more segments as needed. In this way the display of documents would be demand-driven in that a query may only initially return a fraction of the documents in a query, while allowing the user to request more as needed.

[9] Although recent work by Shneiderman [Shn94] suggests how one might, in principle, do so. But his techniques still need work to scale beyond small databases.

Chapter 4

The Interface

In this chapter we explore three Lifestreams interfaces developed for the prototype. We first describe our X Windows client[1], which provides a rich interface to the Lifestreams system. We then describe a command line interface that is suitable for the "lowest common denominator" of computer displays: dumb terminals. Last we present an interface for the Apple Newton PDA, which provides Lifestreams access via minimal (possibly mobile) communication channels.

We begin this chapter by briefly discussing the philosophy behind our interface design and then move on to describe how each interface provides various Lifestreams capabilities. In the next chapter we describe how the X Windows interface is used to accomplish common user tasks. Last we explore directions for future interface development.

4.1 Interface Design

Lifestreams clients may exist eventually on a wide range of computational devices from high-end networked workstations to minimal personal digital assistants, TV set-top boxes and cell phones. A state of the art Silicon Graphics workstation might provide a rich three-dimensional interface to a lifestream, allowing the user to fly through a document collection; a cell phone might provide a voice-activated interface. In this dissertation we lay down no policy concerning the "look and feel" of a Lifestreams client, but each interface should provide a means of using the basic operations. Over time, incremental design and evaluation will result in the development of improved interfaces.

This chapter concentrates on our X Windows client and our attempts to create an effective and satisfying interface for users (see Chapter 6 for more on users' subjective reaction to the interface). While conventional graphical components (such as list

[1] We believe this platform to be representative (with respect to "windowed user interfaces") of today's desktop systems such as the Macintosh and Microsoft Windows.

widgets and scrollbars) may have provided the basic functionality needed to access a lifestream, we have tried to support a "receding stream" visual representation in the spirit of Shneiderman's *direct manipulation* [Shn92] and Nelson's *principle of virtuality* [Nel90].

Direct manipulation involves creating a visual representation of a "world of action." Shneiderman describes direct manipulation as a primary interaction style that presents task concepts visually. By using a visual representation of objects and allowing users to carry out action by pointing and clicking, the user can carry out tasks quickly and observe results immediately. Direct manipulation is good insofar as it is appealing to novices and easy to remember for intermittent users, encourages exploration, and permits high subjective satisfaction (users enjoy using it)[Shn92].

Nelson's concept of "virtuality" is opposed to the usual "metaphorics" [Nel90] approach. Metaphorics is a method of building software based on an implicit comparison to objects or machines in the real world (e.g., the desktop metaphor). Metaphors are useful to a point, but they constrain design in the sense that once the metaphor "has been instituted," every related part of the system has to become part of the metaphor. When designers are forced to add non-obvious functions to the metaphor (e.g., to eject a disk, drag it into the trashcan) users get confused. Nelson argues that "slavish adherence to a metaphor prevents the emergence of things that are genuinely new."

In contrast, virtuality is the construction of unifying ideas that can be embodied in a rich graphic expression that is no mere metaphor for a some pre-existing physical system, but rather, as Nelson argues, leads to new conceptual organizations that have not previously existed. This has been our approach designing Lifestreams: to provide a simple and unified system that is easily grasped by users and not constrained by a real-world metaphor. We now describe the X Windows client and highlight our attempts to incorporate direct manipulation and virtuality into our interface.

4.2 The X Windows Interface

The X Window's Lifestreams interface presents the stream as a receding set of documents (Figure 4.1). Each document is shown as a rectangular region annotated with several text fields: the subject or topic of the document, the creator of the message, and — if the message is a mail message — the sender of the message. Document annotations allow the user to browse the stream and locate specific documents quickly. Future interfaces can do better by presenting actual thumbnail depictions of each document and its contents.

The user can slide the mouse pointer over the stream to "glance" at each document. Figure 4.1 shows a highlighted document and its "glance view." The glance view displays a condensed summary of the document (not to be confused with the summarize operation); in this example the glance view displays the date and time the document was created along with its subject, any mail headers (from, cc), the first couple of lines

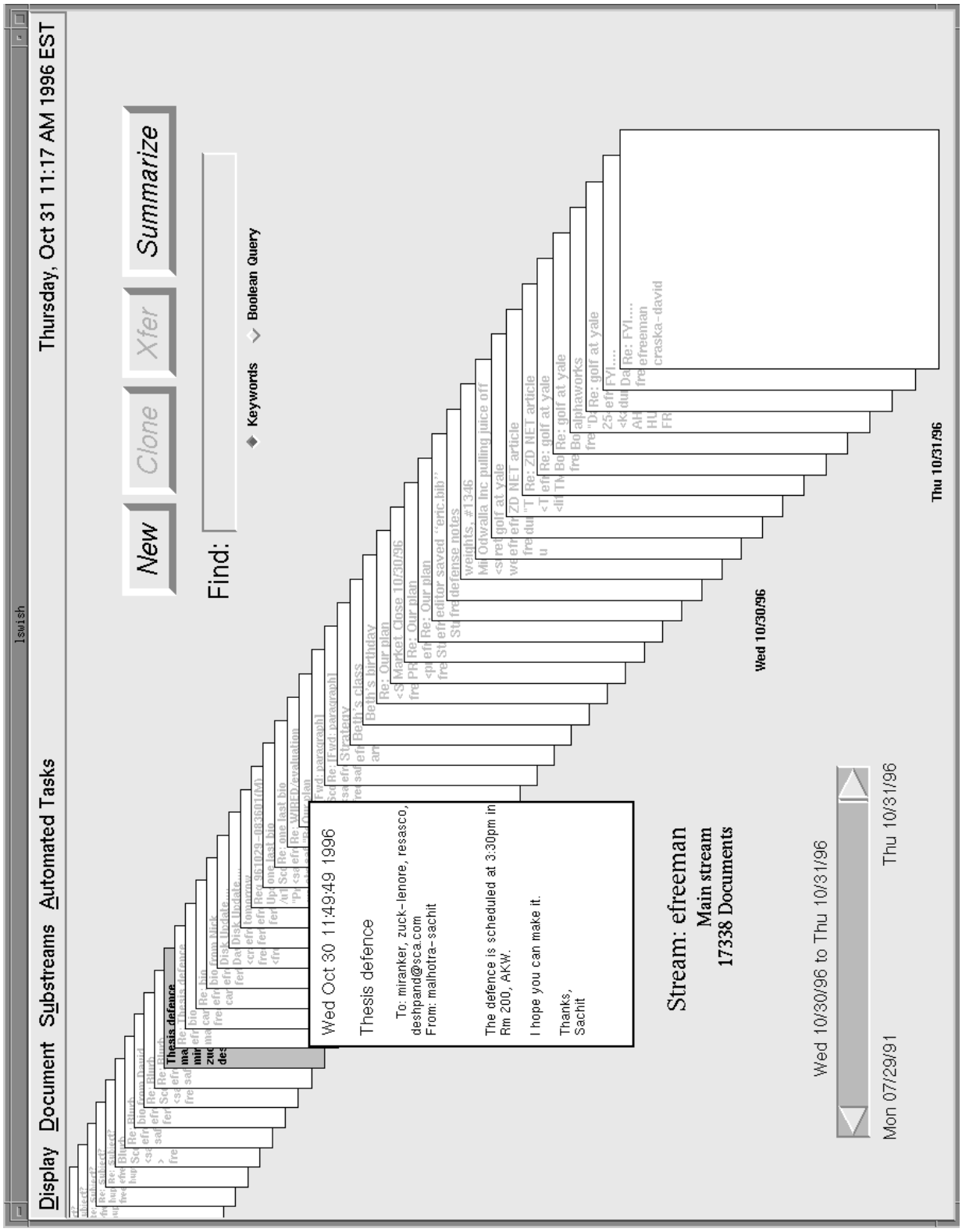


Figure 4.1: The X Windows Interface.

of the document's contents (this last feature is currently supported only for text documents), and an icon that signifies a document agent if one is attached. Glance views help the user browse a stream of documents or read short messages quickly without actually "opening" any documents.

4.2.1 Navigating through Time

The interface displays a "slice" of the stream; on a conventional workstation monitor this slice amounts to roughly forty documents. The creation dates of the documents are displayed alongside the stream. Rather than displaying the creation date of each document, we only display the dates for the first document created each day. This substantially reduces the visual complexity of the display (in the spirit of Tufte [Tuf90]) and makes it easy to group documents into "days of creation." As we have seen previous work has shown that chronological cues are a natural and powerful method of organizing information [Mal83, Lan88b].

Users can navigate through the past via the horizontal scrollbar at the bottom left-hand corner of the interface (enlarged in Figure 4.2). Along the bottom of the scrollbar are the beginning and ending dates of the current stream. In this case the stream includes documents from the end of July 1991 to the end of October 1996. We can see from the display just above the scrollbar that this includes about 17,000 documents from efreeman's main stream (i.e., his lifestream, not a substream). Along the top of the scrollbar we display the inclusive dates of the slice of documents visible within the display. Clicking and dragging the scrollbar slider results in an update of this interval, allowing the user to navigate to a precise time within the stream (we will discuss navigating to the future shortly). The size of the slider is also important. Its size within the scrollbar trough graphically represents the percentage of total documents (within a stream or substream) that are visible.

Figure 4.2: Browsing Back in Time.

We now consider Figure 4.3. The stream is still efreeman's, but here we are focused on the substream "andrew and lifestreams" that contains 102 documents. The scrollbar

date endpoints have changed; the slider now represents a greater percentage of the total documents (forty of 102). We also see that the date and time stamps are grouped more closely together because the substream contains only a few documents from any given day.

4.2.2 Basic Operations

The basic Lifestreams operations — `new`, `clone`, `transfer`, `find` and `summarize` — are provided through buttons and a text-entry box. Clicking on `new` adds a new document to the stream. To clone a document, the user selects a document, and then presses the `clone` button and a copy of the selected document is added to the stream. To select a document, the user presses the second mouse button while the mouse pointer is positioned over the document. Doing so highlights the document and makes the `clone` and `xfer` buttons active. Pressing `xfer` causes a prompt for recipient addresses (see figure 4.4) and then transfers the selected document.

Pressing the `summarize` button applies an appropriate summary to the current substream. As the user creates substreams and "visits" existing substreams the client iterates through available summarizers and evaluates their `test` functions. If only one summarizer can be applied to the stream then the `summarize` button is changed to reflect the available summarizers (we will see an example shortly). If more than one summarizer is appropriate then the client prompts the user when he presses the `summarize` button. Each summarizer results in a document being added to the main stream (and any appropriate substreams). We give examples of summarizers in Chapter 5.

The user can create a substream by typing a search query into the `find` text-entry box and pressing return, as seen in Figure 4.5. The radio boxes below the text-entry box allow the user to select between input methods: keywords and boolean queries. In keyword mode the user enters search terms (such as "lifestreams david") and the system treats them as a series of terms connected with boolean `AND` ("lifestreams david" becomes "lifestreams AND david"). In boolean query mode the user is free to enter arbitrary boolean expressions containing search terms along with `AND`, `OR`, and `NOT`. When the user presses return the substream is computed and the current stream (or substream) is quickly (on the order of one second) replaced by the documents that match the search query.

At any time the user can refocus the display on the main stream or any existing substream. Users can also create substreams in an incremental fashion by using the `find` operation on substreams. A list of substreams and operations is maintained in the `Substreams` menu (Figure 4.6). The first menu option `Remove` deletes a substream and any of its children. Next, the `Your Lifestream` option focuses the interface back on the entire document collection. Last, the menu contains a list of all substreams, which are labeled with their search queries (keywords possibly connected with boolean operations). Selecting a substream focuses the interface on that substream. Nested

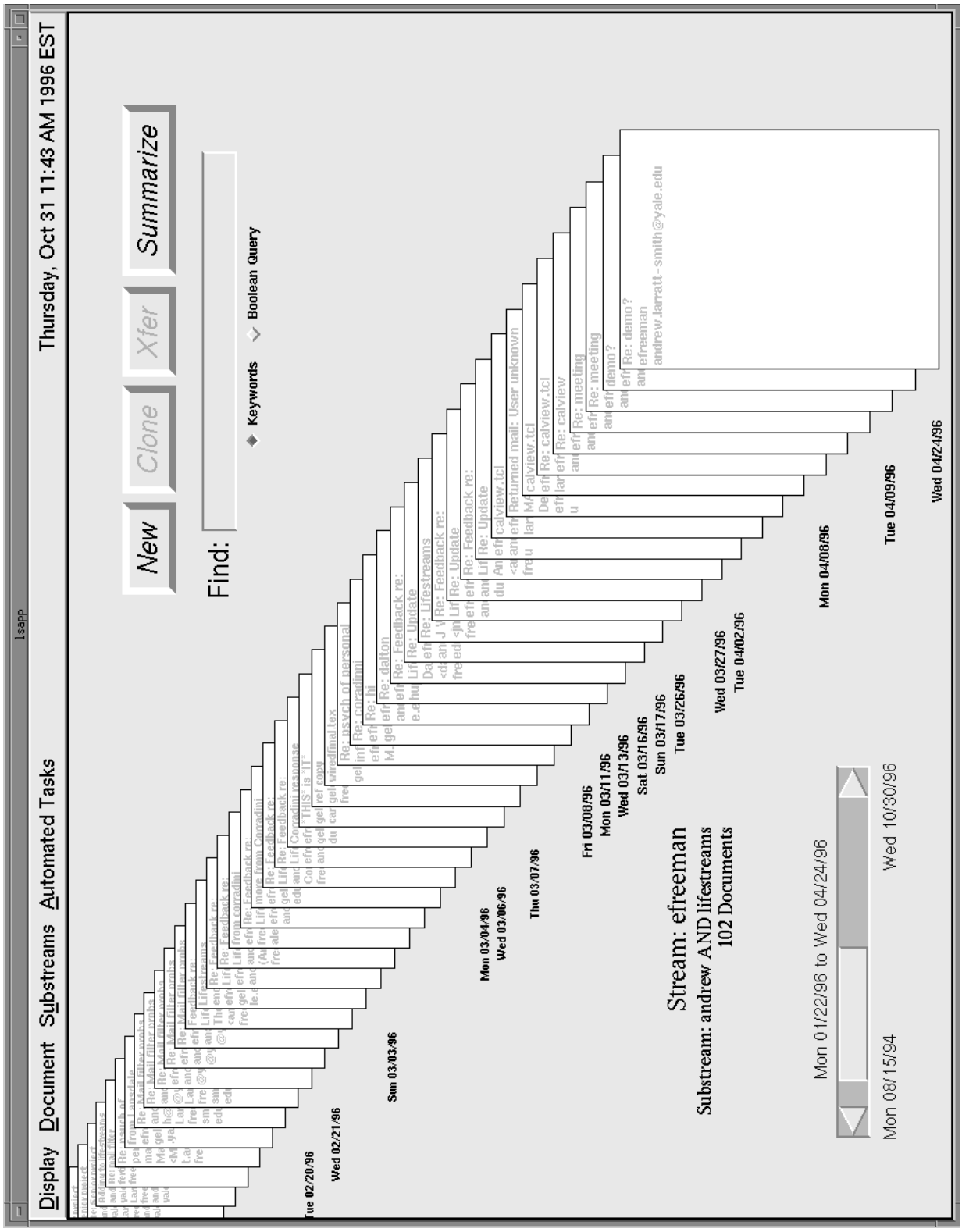


Figure 4.3: Browsing Back in Time with a Substream.

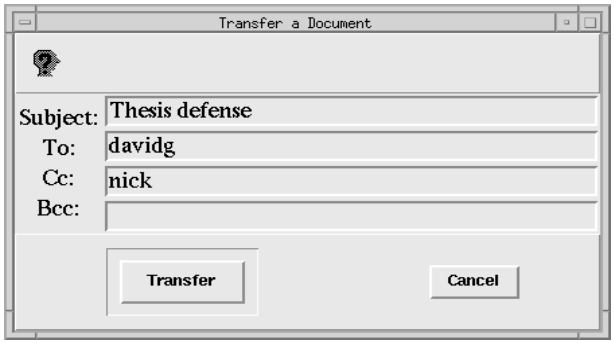


Figure 4.4: Transferring a Document.

Figure 4.5: Using Find.

menu items represent substreams that were created incrementally. In this example, the substream "fertig" was created from the substream "scenarios," which was created from the substream "lifestreams and david," which was created from the entire lifestream. It is important to note that these substreams represent not a hierarchy of information, but a parent/child relationship among substreams. A child substream is a subset of its parent stream, but sibling substreams can overlap in arbitrary ways.

4.2.3 Color and Animation

We use animation to show important events and provide a more convincing "world of action" in Lifestreams. Documents appended to the stream— incoming Internet email, reminders or documents added to a stream by another Lifestreams user—slide in from the left. In the process, the stream is pushed backwards by one document into the past. Newly created documents slide down from the top and assume the front-most position on the stream. Transferred documents slide out to the left. Figure 4.7 shows these actions. The interface allows users to add a sound to each action. Current users use a "swoosh" sound to signify documents arriving on the stream.

Important document states are shown graphically: the borders of unread documents are colored red, while the borders of writable documents are made thicker. Open documents are offset to the side to show that they are being viewed or edited.

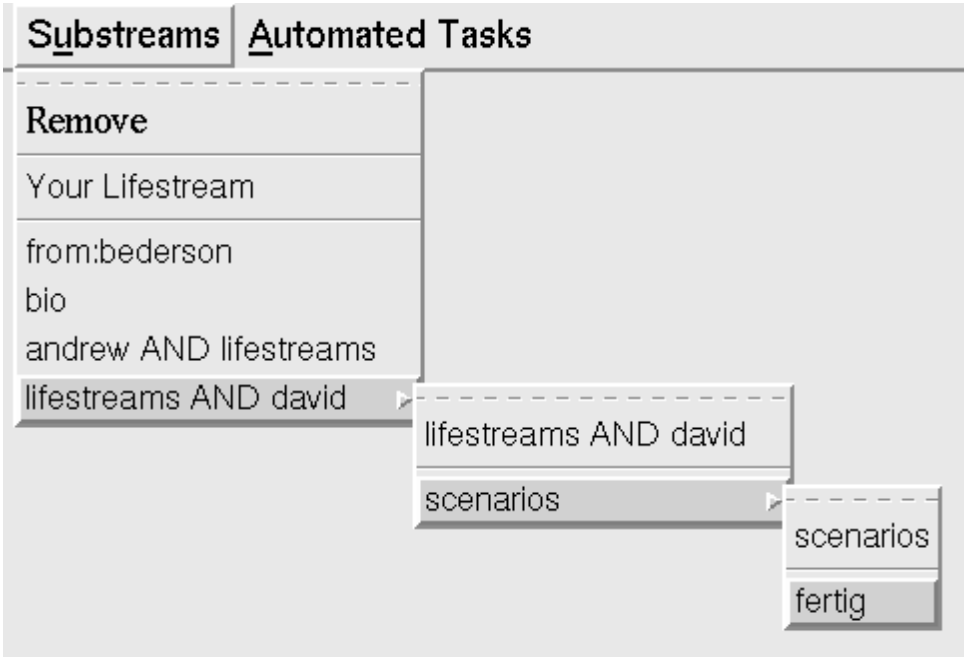


Figure 4.6: Selecting a Substream.

4.2.4 Future Time

By default the Lifestreams interface presents the user with a view of the stream from the present moment receding into the past. As we have discussed, a lifestream also has a future, which (with the X Windows interface) is accessible but usually hidden. The user accesses time-related features of Lifestreams through the clock in the upper right hand corner of the interface. The clock typically displays the current time; it also acts as a menu (Figure 4.8), which allows the interface time to be temporarily reset so the user can observe the future part of the stream. The user can either choose to set the time ahead some predetermined interval (a day, a week, two weeks, etc.) or to set the time ahead in a precise way (via the `Set Time to the Future or Past` menu option) with the calendar dialog box displayed in Figure 4.9. Specifying a future time, either by the calendar of predetermined intervals "refocuses" the interface to include any future documents that have creation dates before the specified time. In Figure 4.10, time was set to the future date of November 30th (one month ahead), and as a result nine new documents appeared on the stream. The division between future and past is marked by the `now` arrow. Also note that the summarize button has changed to reflect the fact that a future summarizer can be applied to the stream (we will explore the future summarizer in Chapter 5).

While visiting the future users can deposit reminder notes or transfer documents to other users (which become future documents on their streams) or can perform any

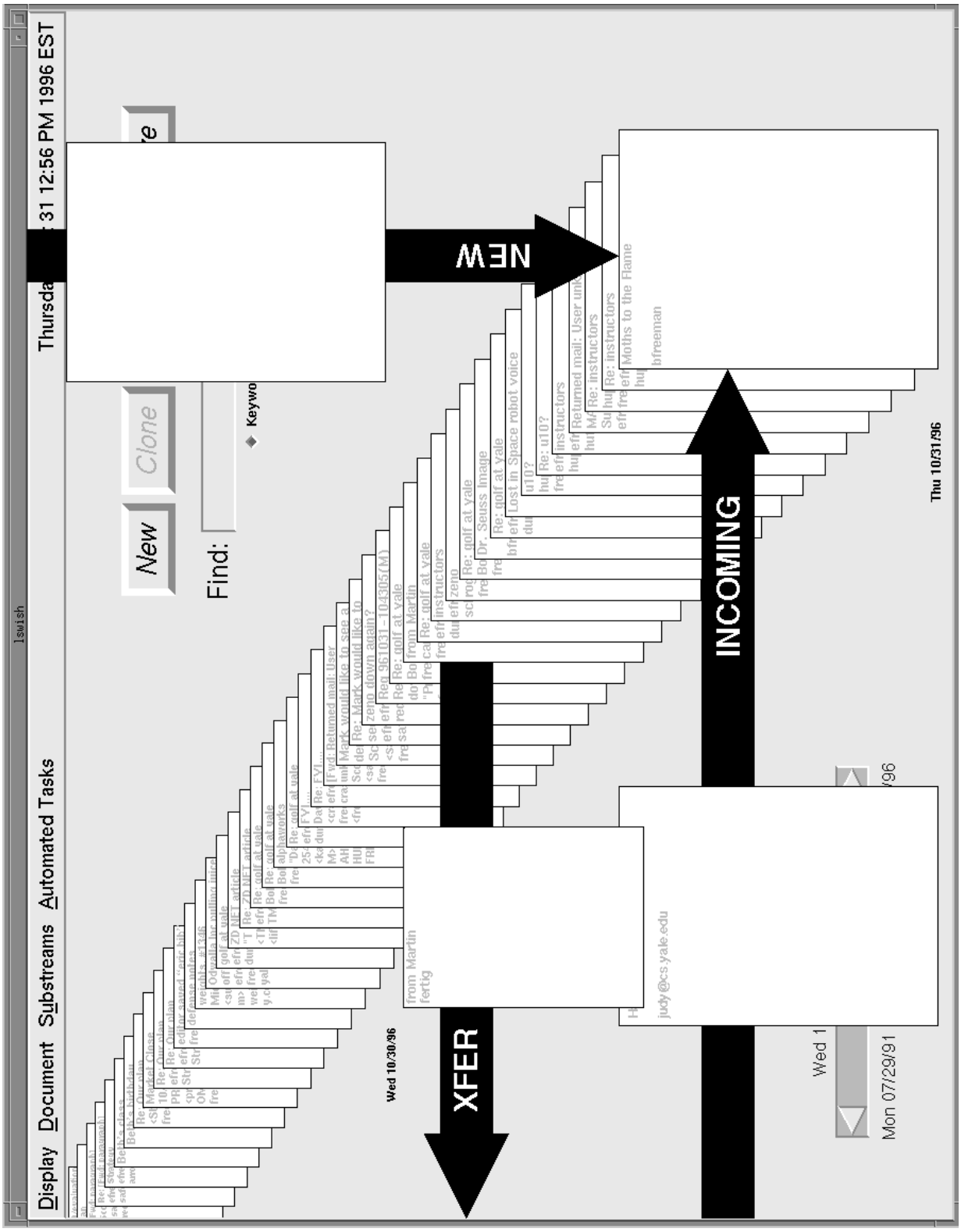


Figure 4.7: Use of Animation within X Windows Interface.

Figure 4.8: Specifying Time.

other operation that is allowed in the present. When the clock is reset to the present these documents disappear, but they arrive on the stream at the appropriate time (when their creation dates roll around).

The user can also specify a past time with the dialog box; this has the effect of scrolling the stream back to that date.

4.2.5 Opening, Viewing and Editing Documents

To see a whole document or edit it the user must first open it. Users open documents by clicking on them with the left mouse button. The document slides to one side to show that it is being viewed or edited. The way in which documents are viewed or edited is largely determined by the choice of a document model.

As we described in Chapter 3, our current system uses the "hypertool model": rather than construct large monolithic applications we build systems out of many small reusable components. This approach has several advantages for Lifestreams. Most importantly, we did not have to reimplement the many existing editors, viewers and other applications our users already accustomed to; they can continue to be used within Lifestreams as helper applications. Helper applications are used as follows: when a Lifestreams client is asked to open a document, it finds an external application that can view or edit the document type and spawns the application to do the job. **xv**, for example, is a common application used to view images; when Lifestreams encounters a document with type "image/jpeg," Lifestreams spawns **xv** to display the

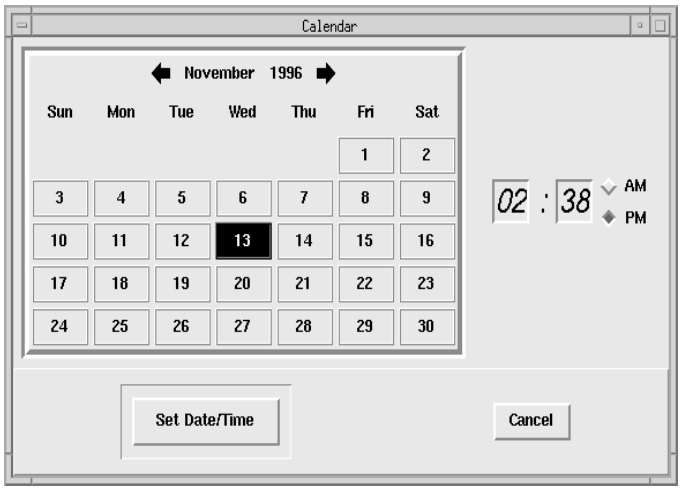


Figure 4.9: The Calendar Dialog Box.

document. This model is used by many existing web browsers to provide a means of viewing web-content that is not supported internally within the browser.

The mapping from types to applications is user-defined. Lifestreams keeps an internal list of user-defined applications that are derived from a "mailcap" file. A mailcap file is a file format standard that was developed for specifying locally-installed facilities for multiple mail formats such as MIME [Bor93a]. Today, mailcap files are used in many multimedia applications, such as web browsers.[2]

Figure 4.11 shows a mailcap file. Each entry specifies a MIME content-type/subtype[3], or a content-type and a wildcard "*" that specifies all subtypes belonging to that type, and then a helper application that can be used to process the MIME document.

Lifestreams as a Window Manager

Figure 4.12 shows a user opening several documents simultaneously in Lifestreams. Each open document is offset to the right[4] as an indication that it is being handled by a helper application. The user has opened documents of several different types and corresponding applications have been invoked to display them: a web url (`netscape`), an image file (`xv`), and an audio file (`audiotool`). When the user closes the application, the respective document slides back into position.

Lifestreams also acts as a window manager: clicking on an open document within the interface causes the corresponding helper application to come to the foreground. This allows the user to maintain a number of open documents and to find documents

[2] Applications on the Mac and PC often provide a user interface for editing the mailcap file.

[3] These types are usually defined in a system-wide file called `.mime.types`, which lists all available MIME types.

[4] This offset is user-defined and can be up, down, left or right.

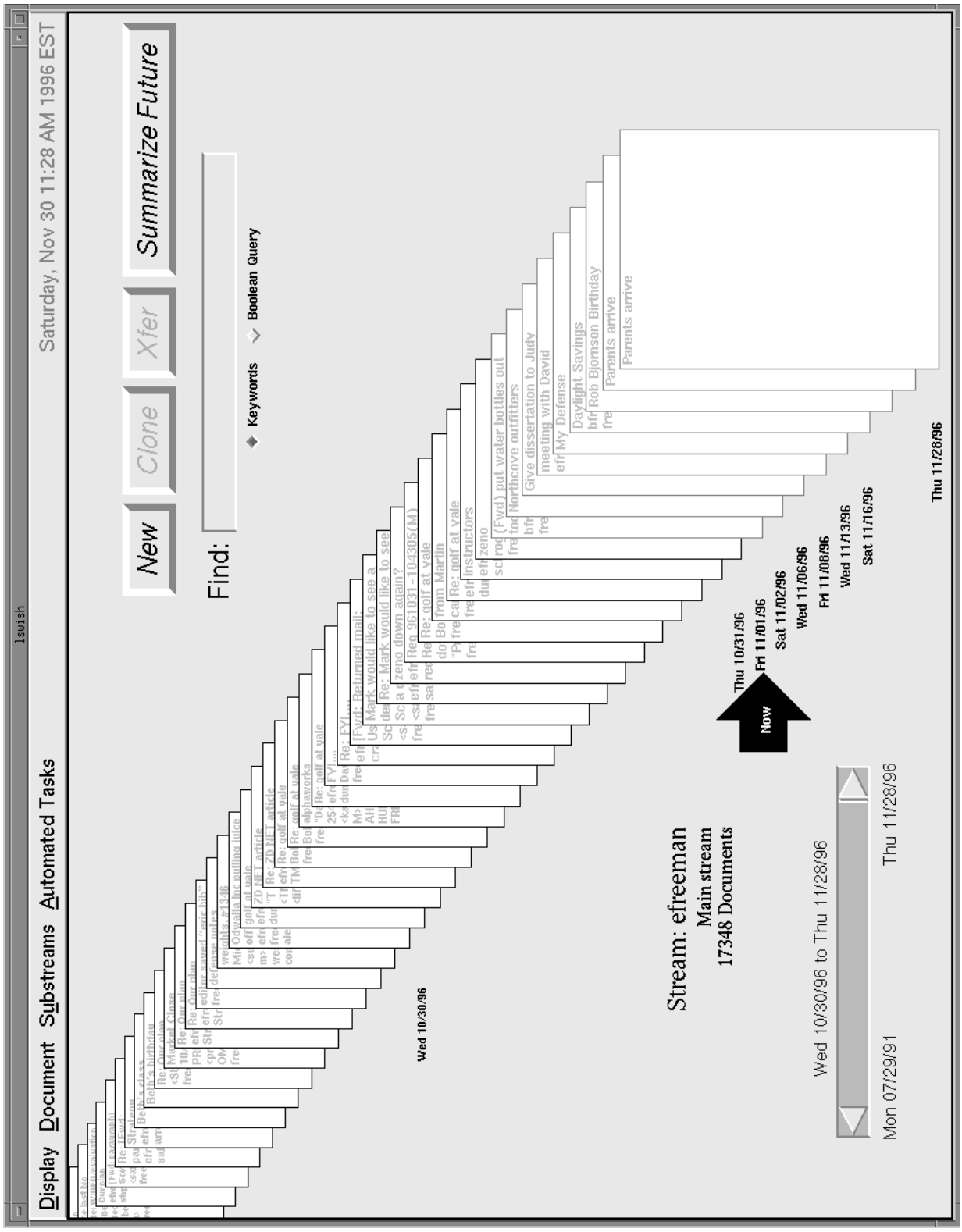


Figure 4.10: Future Documents on Stream.

```
## This is a simple example mailcap file.
## Lines starting with '#' are comments.
#
## This maps all types of audio data (audio/basic, audio/x-aiff,
## etc.) to the viewer 'audiotool'.  Note that '%s' means put the
## datafile name here when the viewer is executed.
audio/*; audiotool %s

## This maps all types of images (image/gif, image/jpeg, etc.)
## to the viewer 'xv'.
image/*; xv %s

## This maps MPEG video data to the viewer 'mpeg_play'.
video/mpeg; mpeg_play %s

## This maps all types of video *other than MPEG* to the viewer
## 'xanim'.
video/*; xanim %s

## This maps all postscript to ghostview and dvi files to xdvi
## and all man pages to nroff
application/postscript; ghostview %s; needsterminal
application/x-dvi; xdvi %s
application/x-troff-man; nroff -man %s

## This maps all PDF documents to acroread
application/pdf; acroread %s

## This maps all tcl scripts to the shell (which will exec them in tclsh)
application/x-tcl; %s
```

Figure 4.11: An Example Mailcap File.

quickly. If a user creates a substream (or views an existing one) that contains an open document the document will be offset in the substream, and clicking on it will move its helper application to the foreground.

In order to write back changes to the server, Lifestreams tracks open documents. When the Lifestreams client spawns (via fork/exec) a helper application to edit a document it adds its process identifier to an internal process table along with the name of a temporary file that is passed to the helper application. When a child process of the client dies (possibly as the result of a user that is finished with the helper application), the client receives a `SIGCHLD` signal and looks for a matching process in its internal process table. If the match is found and the document is writable then the contents of the temporary file are written back to the server. While our current approach is to perform one writeback per user session, the client can be extended to support incremental writebacks by spawning an additional process that communicates with the helper application and performs multiple writebacks directly to the server.

Attributes and Editing

Lifestreams allows users to edit document attributes. Figure 4.13 shows the document attribute dialog box. This dialog box is opened when the user selects a document and then clicks on the third (right-most) mouse button. Through the dialog box users can edit common attributes such as the document's permissions. The user can also assign additional keywords to a document for retrieval purposes. These keywords are added to the document's index at the server.. We will explore their use further in Chapter 5.

Advanced users can also edit the document's MIME type and its attached agents. Figure 4.14 shows the dialog box that is displayed when the user clicks on the `Mime Type` button in the attribute dialog box. The MIME dialog box allows the user to set the document's MIME type to any type defined in the user's `.mime-types` file. Why would users want to change a MIME type? Two examples: when a new document is created the user may want change it to a particular type, or may want to switch between two types when editing two types that have the same representation but different display methods. For instance the user may want to create an HTML document (or in general any document that is specified in a markup language) by editing a plain text document and then changing the type to HTML to render the document. Such operations are usually for advanced users (as is the attribute editing dialog box).

Advanced users may also access document agents through the `Agent` button. Clicking it yields the agent dialog box shown in Figure 4.15. The agent dialog box allows the user to add new agents, to edit (view) the agents of writable (read-only) documents, to read a description of the agent, and to remove agents from writable documents. Editing an agent (by selecting an agent and clicking on the Edit button) brings up a editor for that agent. The editor environment for agents is currently primitive but usable. In the editing window the user edits four "fields" of text: its name, its description, it type, and finally the source code for the agent. Using this editor the user can iteratively

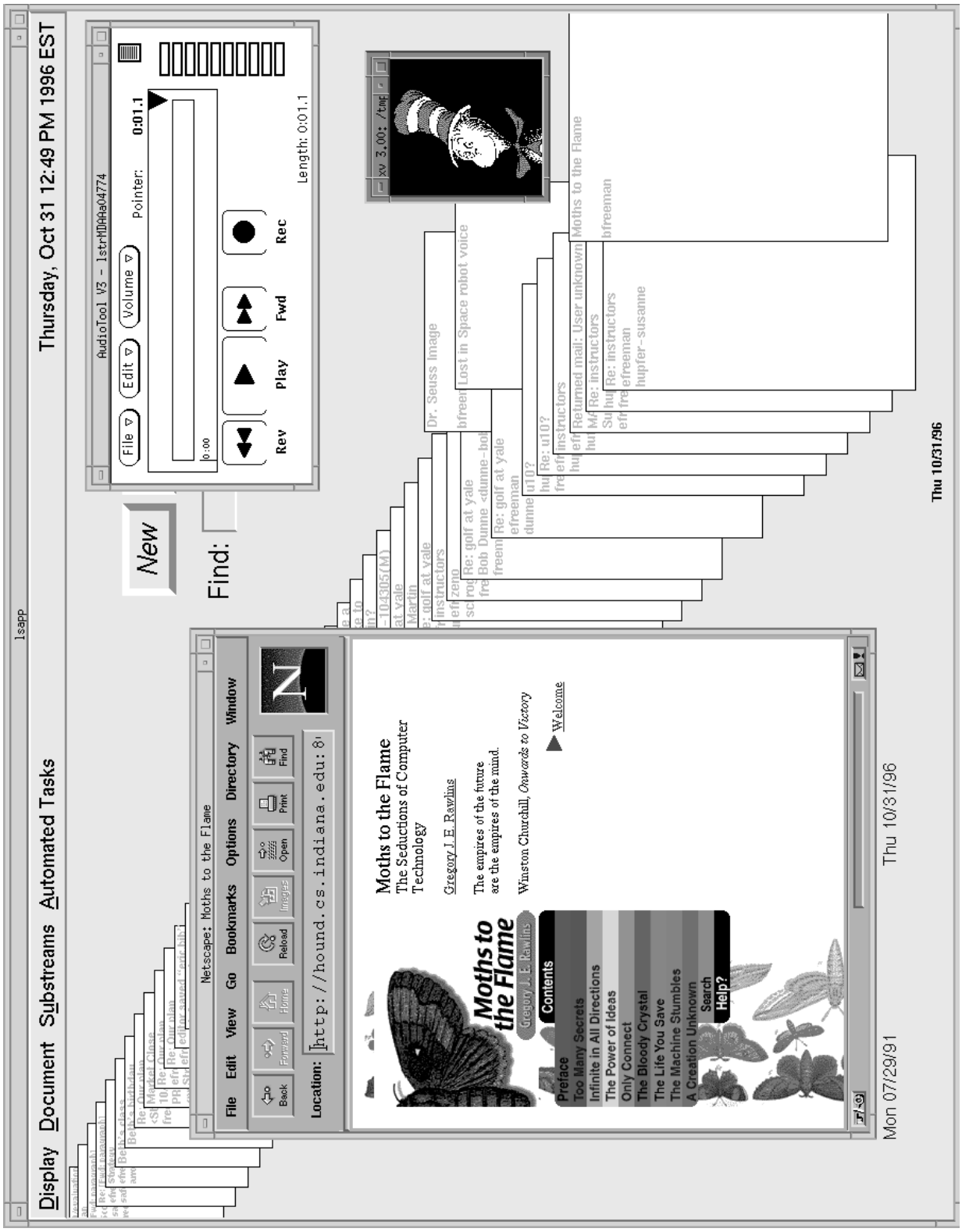


Figure 4.12: The X Windows Interface Handles MIME-typed Documents.

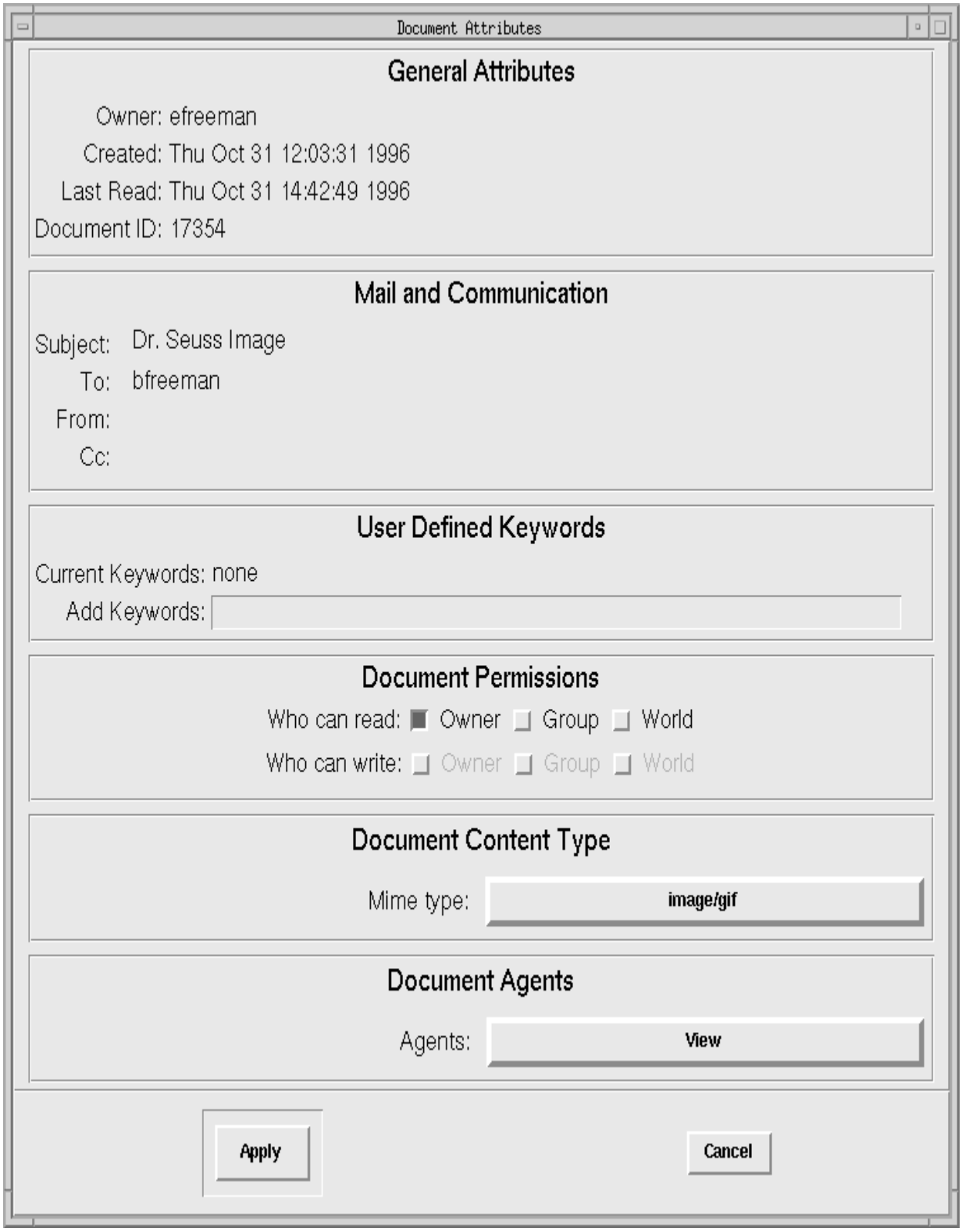


Figure 4.13: Editing Document Attributes.

Figure 4.14: Choosing a Mime Type.

Figure 4.15: Editing an Agent.

write agent code and test it within the system (by opening the document which causes the evaluation of the agent).

4.3 The Command-line Interface

Our work on the command-line client was driven by user demand. While the initial user group was content using Lifestreams on workstations at the office, we soon found it frustrating not to be able to access our streams from home. There was additional motivation to do so, as some users prefer a command line interface: witness the number of users who use `vi` over graphical word processors, or who use command line mail programs rather than graphical mail programs such as `xmail`. Many of us commonly use the UC Berkeley mail interface and it seemed a good base for building a command line interface (CLI) to Lifestreams.

The Lifestreams CLI (Figure 4.16) looks like the UCB mail application, however it uses a "lifestream" as a basis for storage rather than a spool file. Here, the user has asked the CLI to display the last twenty documents in his stream. Each document is listed by a document identifier, which may be preceded by a number of one-character descriptors (e.g., `W` = *writable*, `N` = *new*, `U` = *unread*, `A` = *agent attached*). Next, the sender of the document is displayed, followed by the date and time the document was created or added to the stream, and a subject (if one exists). Documents with a blank sender field were created by the owner of the lifestream. Also note that while this slice of the stream contains mostly electronic mail[5], there is also a TR draft and a URL created by an agent on the stream.

4.3.1 Viewing/Editing Documents

Like the X Windows client, the CLI supports helper applications[6]. If the CLI is being used from an X Windows console then the appropriate helper application is launched to view or edit a document. If, on the other hand, the CLI is running from an ASCII terminal, an error is reported that the helper application can't be opened or, in some cases, an alternative helper application is used. For example, when a document of type `application/x-url` is opened, the CLI either executes an X Windows web browser (usually Netscape), or, if the user is using an ASCII terminal, executes `lynx`, the text-based web browser.

4.3.2 Substreams

`Lsmail` provides direct support for substreams through `find` and four additional commands for navigation (as provided by menus in the graphical interface): `substreams`, `substream`, `remove`, and `up`. The `inc` command (which, in UCB mail, includes new documents that have arrived since the user last checked) has also been implemented in the CLI to provide the same functionality.

[5] And if fact most streams do because we tend to generate more electronic mail documents than any other type.

[6] The UCB mail program only supports text mail messages, which are commonly displayed with the `more` program

```
Lifestreams (YALE) version 0.922 beta  Type ? for help.
Eric_Freeman@pythagoras (your lifestream): 4507 documents 50 unread
& h $
  4486  hupfer-susanne@cs   Wed Apr 19 17:55 95   No Subject
  4487  sullivan@fa.disne   Wed Apr 19 19:22 95   weights, #1013
  4488  sholden@cod.nosc.   Thu Apr 20 07:39 95   Newton_User_Groups_v1.11
  4489  fertig@cs.yale.ed   Thu Apr 20 12:21 95   Size of key?
  4490  fertig@cs.yale.ed   Thu Apr 20 12:44 95   Public key on casper
  4491  fertig@cs.yale.ed   Thu Apr 20 12:54 95   Re: testing
  4492  fertig@cs.yale.ed   Thu Apr 20 12:54 95   Re: testing
  4493  hupfer-susanne@cs   Thu Apr 20 13:12 95   No Subject
  4494  Netscape Agent      Thu Apr 20 13:13 95   http://www.voyagerco.com
  4495  eperry@SUNED.ZOO.   Thu Apr 20 15:03 95   progress
  4496  eperry@SUNED.ZOO.   Thu Apr 20 15:46 95   Re: progress
W 4497                      Thu Apr 20 16:02 95   TR draft
  4498  eperry@SUNED.ZOO.   Thu Apr 20 16:07 95   Re: progress
  4499  eperry@SUNED.ZOO.   Thu Apr 20 16:24 95   Re: progress
  4500  eperry@SUNED.ZOO.   Thu Apr 20 16:33 95   Re: progress
  4501  Quote-O-matic       Thu Apr 20 17:00 95   Portfolio for 4/20/95
  4502  PIESYSOP@AppleLin   Thu Apr 20 17:21 95   Re: Signature Register
  4503  hupfer-susanne@cs   Thu Apr 20 17:21 95   No Subject
  4504  sullivan@fa.disne   Thu Apr 20 19:07 95   weights, #1014
N 4505  fertig@cs.yale.ed   Fri Apr 21 10:43 95   Re: PGP
N 4506  fertig@cs.yale.ed   Fri Apr 21 10:53 95   No Subject
```

Figure 4.16: The Command-line Interface.

In Figure 4.17, we present a user session that includes these commands. The transcript begins as a user is viewing his main stream (all 4500 documents). The user then issues the command:

```
find lifestreams and dissertation
```

This results in the creation of a substream containing sixteen documents from the entire stream. The user then lists the documents with the "`h $`" command. For all practical purposes the user can treat the substream as if it were a general mail spool. The user then issues the `substreams` command, which displays all existing substreams. Incrementally created substreams are indicated through indentation and the current stream is preceded with a `*`. The user then removes the substream with the `remove` command. The `remove` command defaults to the current substream, but can also be given an argument to explicitly remove another substream. After removing the substream the `remove` command returns the user to the parent substream, in this case the main stream. The user then issues the `substream` command, moving the user to the "shneiderman" stream. Issuing the `up` command moves the user to the parent substream without removing the child. Last, the user issues the `inc` command, which checks the current stream or substream for new documents (in this case there was one additional document appended to the stream).

4.3.3 Time

The command-line client time can be set via the `settime` command. This command requests the time from the user. Users are allowed to enter dates via a flexible format based on the `getdate` parser developed by Steven M. Bellovin. As an example, Figure 4.18 shows a user setting the time to next Tuesday at 8am. Note that the "`h $`" command now displays a number of future documents.

The user can reset the time to the present with `settime now`.

4.3.4 New, Clone and Transfer

The command-line client supports `new`, `clone` and `transfer`. It also allows the user to type `mail` along with a recipient list, which automates `new` and `transfer` for the user. As with the X Windows interface all of these commands can be carried out in future time.

4.3.5 Summaries and Personal Agents

The command-line client supports both summaries and personal agents, however more work needs to be done to provide compatibility with the X Windows client. Porting a summarizer or personal agent from the X version currently requires changes to the client's source code (rather than just a change to a "defaults" file as in the X version)

```
& find lifestreams and dissertation
Eric_Freeman@pythagoras (lifestreams and dissertation): 16 documents 1 unread
& h $
   0    fertig              Fri Nov 03 15:28 95   Re: Hiding documents in Life
   1    Eric_Freeman        Mon Nov 13 16:24 95   LS evaluation
   2    David Kaminsky      Tue Nov 14 09:14 95   lifestreams (LS)
   3    Eric_Freeman        Tue Nov 14 10:51 95   Re: lifestreams (LS)
   4    David Kaminsky      Tue Nov 14 12:28 95   lifestreams (LS)
   5    Eric_Freeman        Tue Nov 14 13:12 95   more from Kaminsky
   6    ben@cs.UMD.EDU      Wed Nov 15 10:40 95   Re:  Committee
   7    Eric_Freeman        Wed Nov 15 10:54 95   from ben
   8    ben@cs.UMD.EDU      Wed Nov 15 11:05 95   Re:  Committee
   9    Eric_Freeman        Sun Dec 03 16:29 95   draft of letter to Franklin
   10   fertig              Sun Dec 03 16:39 95   Re: draft of letter to Frank
   11   Eric_Freeman        Sun Dec 03 16:46 95   Re: SIGMOD
   12   Eric_Freeman        Sun Dec 03 16:53 95   Re: SIGMOD
   13   franklin@cs.UMD.E   Mon Dec 04 10:27 95   Re: SIGMOD
 N 14   Eric_Freeman        Mon Dec 04 10:30 95   Re: SIGMOD
   15   Eric_Freeman        Tue Dec 05 19:14 95   Re: Lifestreams evaluation
&
& substreams
  0. All documents in your lifestream
     1. franklin
     2. shneiderman
     3. ben
     4. netscape and bookmark and hotlist
         5. sigmod
         6. franklin
         7. hcil
     8. stock and service and gvil
     9. chi
     10. paul
        11. Gianattasio
     12. ethan and lifestreams
     13. Mikrotec
    *14. lifestreams and dissertation
& remove
Eric_Freeman@pythagoras (your lifestream): 4509 documents 51 unread
& sub 2
Eric_Freeman@pythagoras (shneiderman): 27 documents 0 unread
& up
Eric_Freeman@pythagoras (your lifestream): 4509 documents 51 unread
& inc
Eric_Freeman@pythagoras (your lifestream): 4510 documents 52 unread
&
```

Figure 4.17: Substreams in the CLI.

```
& settime
Enter a time [? for help]: next tuesday 8am
Set time to Tue Jun 25 08:00:00 1996? y
Setting time to Tue Jun 25 08:00:00 1996.
efreeman@pythagoras (your lifestream): 13580 documents 28 unread
& h $
   13559 Jon Bennett <jonb   Thu Jun 13 16:34 96   Re: Eric Freeman's Home Pag
   13560 support@devtools.   Thu Jun 13 16:40 96   Re: Cafe Studio?
   13561 efreeman            Thu Jun 13 18:04 96   Re: Eric Freeman's Home Pag
   13562 efreeman            Thu Jun 13 18:06 96   Re: Cafe Studio?
   13563 hupfer-susanne      Thu Jun 13 20:11 96   raccoons
   13564 PRICES <prices@Bo   Thu Jun 13 20:18 96   Market Close 06/13/96
   13565 The c|net newslet   Thu Jun 13 23:00 96   c|net Digital Dispatch Vol.
 N 13566 sullivan@huey.dis   Fri Jun 14 00:18 96   weights, #1252
   13567 freeman-elisabeth   Fri Jun 14 09:20 96   msg
   13568 freeman-eric        Fri Jun 14 09:27 96   editor saved ``processmail'
 N 13569 freeman-eric        Fri Jun 14 09:27 96   editor saved ``/tmp/lstrMBA
   13570                     Fri Jun 14 09:44 96   Yale Club
   13571 efreeman            Fri Jun 14 09:50 96   Como
   13572 efreeman            Fri Jun 14 10:30 96   Re: Hello
 F 13573 efreeman            Fri Jun 14 14:00 96   Jones/McRae meeting
 F 13574 efreeman            Sun Jun 16 08:00 96   Father's Day
 F 13575 efreeman            Mon Jun 17 08:00 96   Dermatology Appointment
 F 13576 bfreeman            Mon Jun 17 09:28 96   Reminder: Saturn Car Clinic
 F 13577 efreeman            Mon Jun 17 19:00 96   Saturn New Car Clinic
 F 13578 efreeman            Sat Jun 22 13:00 96   Beth/Susanne talk at SCSU
 F 13579 efreeman            Mon Jun 24 08:00 96   Deposit refund due
& settime now
Setting time back to``now."
efreeman@pythagoras (your lifestream): 13573 documents 21 unread
&
```

Figure 4.18: Time in the CLI.

```
& remind
Enter a time for the reminder [? for help]: next wednesday 10am
Create reminder on Wed Jun 26 10:00:00 1996? y
Subject: meeting with David
Meet with David about the dissertation.
.
creating reminder on Wed Jun 26 10:00:00 1996...done.
```

Figure 4.19: Reminder personal agent in the command-line client.

and also to the agent if it makes use of the Tk graphics libraries (because we no longer have a window display). One personal agent supported is the "reminder" agent (Figure 4.19). Like the X version it first prompts for a date/time and then for a message, which is then added to the future. The user can specify other recipients for the reminder and a `xfer` is automatically done to their streams.

A couple of the summarizers are also supplied (the default summarizer and the LaTeX summarizer). These are accessed by typing `"summarize"` at the command line.

4.4 The PDA Client

We have also implemented Lifestreams on the Apple Newton [Com93]. This work was first described in [Fre95]. The Newton presents an interesting set of constraints for developing a client. It has a small (240x360) monochrome graphics display, limited memory (160K of free heap and 2M of flash card storage), limited communications bandwidth (2400 baud), and pen-based input.[7] Certainly we could not develop a full-blown version of the X Windows client; however our initial thoughts were to develop a scaled-down version. On closer examination, the Newton suggests intriguing implementation strategies because of several similarities between the Newton operating system and Lifestreams. Like Lifestreams, the Newton:

- Does not use a traditional hierarchy for managing information. Instead it uses a simple folder-based scheme along with...
- A content-based FIND capability as its primary method of locating information.
- Replaces the traditional filesystem with a persistent storage model.

Rather than attempting to integrate Lifestreams into the existing Newton framework—that is, replacing the Newton's static folders with substreams, reworking the built in

[7] These are the specifications for Newton devices at the time of writing

notepad to load its notes from the Lifestreams server rather than the resident "notes system soup," overriding the Newton FIND function to create true substreams, etc.—we've developed the Lifestreams client as a standalone Newton application, while in the process attempting to envision the look and feel of an integrated version. To that end, we've taken some liberties and overridden some of the Newton's default behaviors.

4.4.1 Communication

We've developed a proxy application to sit in between the client and the server because neither the RPC nor the TCP communication protocols are supported in the Newton operating system. The proxy takes requests via ASCII commands from the Newton. The proxy then performs these requests by making RPC calls to the server and returning the results to the Newton in mixed ASCII and binary formats. The Newton accesses the proxy by either dialing into a terminal server via modem or through a tethered connection to a workstation. After connecting, the Newton then connects (via telnet) to a specific port on a proxy host and begins communicating with the proxy. This indirect scheme of telneting to a proxy host allows the user to access his lifestream anywhere on the network, as long as he has access to a local machine that supports TCP and the telnet protocol.

4.4.2 The Newton Interface

Figure 4.20 displays the Newton interface. The user is initially greeted with a connection dialog that allows him to specify the name of his stream, the name of his Lifestreams server (which in this case is actually the proxy, not the server), and the connection method: modem, a direct tether connected to a UNIX workstation, or a wireless connection (not currently supported). Like most Newton applications, Lifestreams contains a folder icon at the top of the interface. The folder normally contains a set of predefined categories for filing. We have overwritten this behavior and replaced the standard folder name with the user's stream name. We will see how the folder functions shortly. A clock in the form of a small circle at the bottom of the interface is also a standard feature of Newton applications. Its default behavior is to display the date and time when tapped on; we will see how it is used with Lifestreams shortly. Also included in the interface is an information button (that displays information about the application and its author) and the essential Lifestreams buttons: NEW, CLONE, XFER and FIND[8], followed by the application's "close" button (the button with an X).

To access a lifestream the user taps on the `Connect` button and the Newton attempts to connect. The application then uses a built-in configuration script to attach to the workstation or terminal server and issue a telnet command to connect to the proxy.

[8]Because the Newton interface was implemented early on in the Lifestreams work (and not maintained over the course of the project) the summarize function was not included.

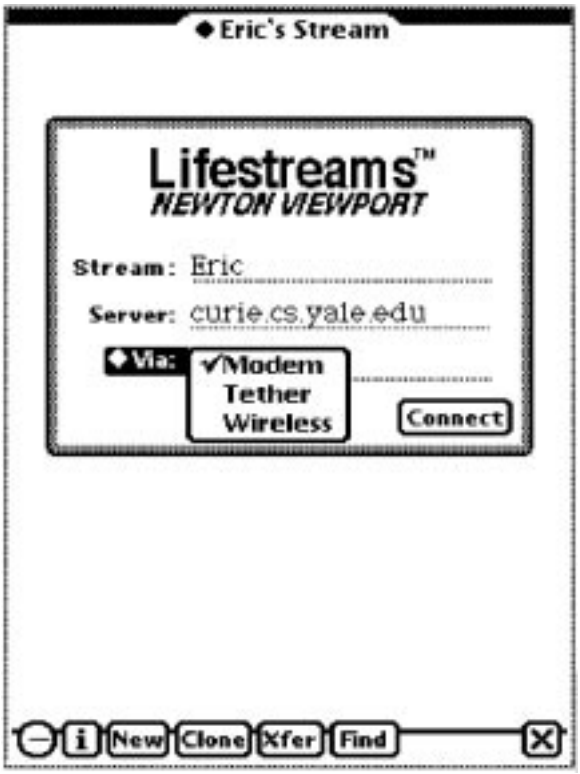


Figure 4.20: The Newton Interface.

After successfully making contact with the proxy, the proxy retrieves an updated list of stream documents and sends this list to the Newton. The client then interacts with the server by issuing ASCII commands and receiving data back. The issuing of these commands is usually user-driven: the user interacts with the interface, and behind the scenes the application issues commands to the proxy to satisfy requests.

Figure 4.21 shows the interface just after the Newton has successfully connected to a stream. As you can see, we display the list of stream documents in an "overview." On the Newton an overview can be obtained by pressing the overview button (a part of the Newton form factor, not visible in screenshots). The overview lists all the documents within the current application. As with any Newton overview the user can scroll down to see more documents (older ones, in this case) or tap a particular entry to retrieve the corresponding document.

Figure 4.22 displays a document that has been retrieved from the Lifestreams server. Once a document is displayed the user can scroll through it with the Newton's built-in up and down arrows. If the document is writable the user can use the Newton's hand recognition input to alter the document. Tapping the overview button redisplays the list of documents in the stream. Our implementation falls short of true Newton

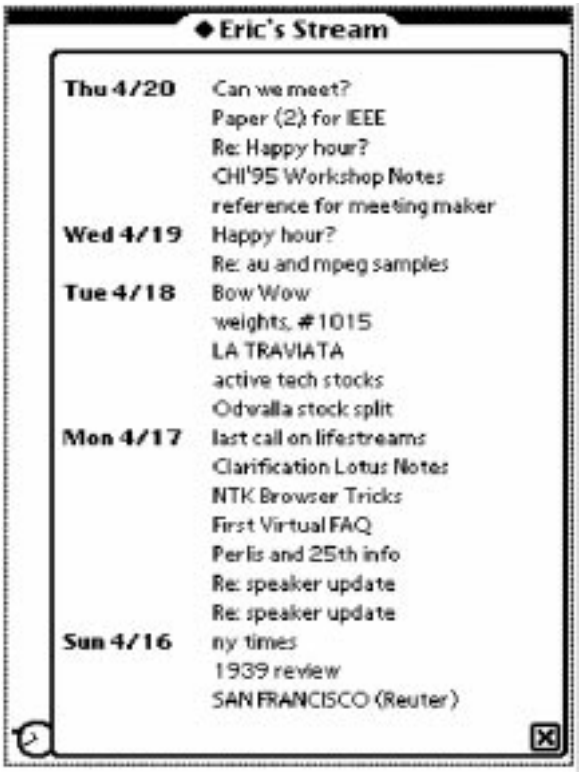


Figure 4.21: Documents in a stream are displayed as an overview.

functionality here, as the up and down arrows only scroll within the document. To be consistent, the client should be able to scroll to the next or previous document as in the built-in Newton notes application. Notes).

Also in Figure 4.22 we see how the folder button has been replaced by our own flavor of folders (substreams). Our folder button activates a popup list that contains the name of the current stream and its associated substreams. Choosing a substream loads the document list associated with the substream (usually a much shorter list of documents than appear in the stream) and displays them in the overview. The label of the folder button is also updated to display the name of the substream. By using the "remove substream" option, the current substream can be removed at any time.

We have also changed the behavior of the clock button in the status bar (see Figure 4.23); when it is tapped a date/time picker opens instead of the normal time glance. As in the X Windows interface, the client application maintains its own "system time" and allows the user to alter this time by dialing to the future or the past (and back to the present). In any case, the client sets its clock to the user specified time. The "system time" is then used when the client issues commands to the proxy/server.

The operation of NEW and CLONE is straightforward: NEW replaces the currently

Figure 4.22: Substreams are displayed in place of folders.

displayed document with a clean slate. CLONE duplicates the current document and allows the user to begin editing the new copy. In either case, when the user finishes editing the document (e.g., when another document is retrieved or the user exits from the CLI), Lifestreams first stashes away the latest changes by saving the document back to the server. In this way, storage is transparently handled.

XFER displays a list of stream addresses from your current server (Figure 4.24). By tapping on the name of a stream the current document is then transferred to that stream.[9]

Similarly, FIND is supported in a simple form on the Newton. Tapping the FIND button displays a dialog that prompts for a boolean query (Figure 4.25). Clicking on `Find` issues a request to the server to create the substream. On the Newton the substream is added to the list of substreams, the document list is loaded, and a new overview is displayed.

While the Newton client is functional and useful, there is still much that can be

[9]The implementation of the Newton client at the time of writing lags behind the X Windows implementation, which builds its transfer operation on top of SMTP. The Newton client still uses a earlier intra-server communication protocol.

Figure 4.23: The clock glance has been replaced by a date/time editor.

done to improve it. We've already mentioned the lack of support for summaries and the active update of live substreams. There are other aspects of Lifestreams that are not supported in the Newton. For instance, the lack of support for MIME-typed documents. The Newton client currently handles only one document type, that of plain text, the most common type. There is no reason that the current Newton version could not handle a variety of text, image, and audio types. By defining a MIME type for the Newton's internal document type we could also store, search, and transfer Newton notes, business cards, and schedules within Lifestreams.

One last aspect of Lifestreams not represented in the Newton is the use of agents. Currently Tcl/Tk has not been ported to the Newton, and doing so would require significant resources. Given this lack of support there is no clear path to supporting agents directly on the Newton in the near future, although it is possible the proxy could be extended to help with agent functions.

With substantial effort it is conceivable that Lifestreams could be integrated into the Newton in a practically seamless fashion. Changes to the Newton based on Lifestreams would also be interesting and would unify many of the disparate aspects of Newton applications. For instance, there were criticisms when the Newton was first released

Figure 4.24: The Transfer dialog.

Figure 4.25: The Find dialog.

that no hierarchical method of data storage was available. Adding substreams would probably be a more satisfying approach. The adoption of our flexible time scheme would also be interesting, as the multistep process of going to the dates application on the Newton would disappear. For those who prefer the native Newton calendar interface, there is no reason this can not be used as an alternate view of a stream.

4.5 Summary

We have presented three Lifestreams clients running on platforms with different capabilities. The Newton client is a proof of concept. The X and CLI clients have been well used, maintained and have played a valuable part in the incremental design and evaluation of the prototype. We will present our evaluation of the interface in Chapter 6; however we briefly point out directions for future work here.

As we stated in the beginning of this chapter we consider our interface design the first of a series of Lifestreams interfaces. Future interface work must explore improving

the information density of the interface, handling large document collections (more than 50,000 documents), a better method for managing substreams, and continue to explore alternative methods for displaying streams.

Information density can possibly be improved in several ways. One method may be to foreshorten the display of the stream so that less screen space is taken up (and wasted) by the stream display. It would also be useful to allow the display of more many substreams at once. A zooming interface (such as Pad++ [BH94]) could also allow the user to zoom in and out or even fly over the stream.

We have also suggested (in Chapter 3) handling large collections of documents through techniques similar to database cursors. Other techniques such as Shneiderman's starfield display [Shn92] use "sliders" to narrow a large number of documents down to a few.

Substreams can most likely be improved by displaying them graphically rather than storing them in a menu. Since users typically remove all but a few substreams, a least-recently-used policy may be used to automatically remove substreams. In addition, a method has been suggested that would place a "calling card" for each substream on the stream so that they are can be found via `find` [CFFG96].

We will explore a few calendar-like methods of displaying a stream in the next chapter. Future work should continue to explore alternative views of the stream including stream access through non-conventional devices such as telephones and TV displays.

Chapter 5

Common Tasks

We now examine Lifestreams in the context of its use and describe how common user tasks are accomplished within the prototype. Our intention here is to survey the landscape of tasks mentioned in Chapter 1 and convey a sense of how they are accomplished using Lifestreams. In the next chapter we will take a more analytical look at the prototype and present qualitative and quantitative data from its actual use.

We begin by first examining the common filing and finding practices of Lifestreams users. We then examine more specific tasks including contact management, calendar applications, and web bookmark management. We take most of our examples from the practices of actual users; however in a few examples we will suggest how Lifestreams could conceivably be used in other contexts (such as in the financial domain where expecting users to use a prototype system for their finances was unreasonable). Along the way we will also describe task automation and system extensions that were developed via agents.

5.1 Finding and Filing Practices

Rather than using filenames and folders, Lifestreams users rely on attribute and content-based "substreaming" to accomplish the tasks of organization and retrieval (or "filing and finding" as it is called in the desktop world). Users tend to organize information within Lifestreams in three different ways that all entail the use of `find` and substreams.

The most common use of the find operation is to create a temporary document collection that can be browsed for a piece of relevant information. A good example of this is searching for a document on a particular topic, for instance the "last draft of the Lifestreams SIGMOD paper." Such queries are typically accomplished by performing a `find SIGMOD and Lifestreams` that results in a substream with small number of documents[1]. These substreams can be visually browsed for the intended document

[1] Our user testing showed that over half of the find operations resulted in substreams with less than 25 documents (see Chapter 6).

or, if the number of documents is large, further searched to locate the document (the user may also make use of chronology to locate the document, even if the substream is large).

In this way Lifestreams allows browsing and searching to be intertwined: the user filters his (possibly large) information collection down into a narrow context in which to search for specific documents. From there the user may further narrow the search again by using find (and so on). Work by Gopal *et al* [GKM95] have noted that these two paradigms (browsing and searching) have been supported separately by previous information systems and have argued that by allowing access to the two capabilities *all the time*, users are given a more powerful tool for finding information. Substreaming supports this combined paradigm naturally: the user can browse his current stream or substream, search, browse again, further refine his search (through incremental substreams), browse again, back up, search again, and so on.

When users create substreams in this manner they typically destroy them after they've located (and possibly cloned or xfer'd) the document. Our evaluation in the next chapter will examine this topic further.

Another way users commonly use substreams is to maintain persistent collections of documents. For instance if the above SIGMOD draft was developed over the long-term, the user might allow the "SIGMOD and Lifestreams" substream to persist (rather than immediately removing it) so that it continues to collect new documents as they are added to the stream. This style of organization directly mirrors what Barreau and Nardi [BN95] call "working" information — information that is typically relevant to an ongoing project or task and is important to the user over a time period of weeks or months (we will explore these classifications further in Chapter 7). In this case the substream of "working" information for the SIGMOD paper might include drafts of the paper as well as email exchanges between the author and editors. Barreau and Nardi found that users often prefer to keep working information in an easily accessible "location." Here Lifestreams shows a bit of the flavor of a location-based system, but with a few differences: First, information in persistent substreams still remains accessible to other substreams. Likewise, at any time the "SIGMOD and Lifestreams" substream can be further refined by creating an incremental substream (say by `find "comments or feedback"`). Last, substreams automatically capture new information as it is added to the stream (so the user doesn't have to explicitly file new documents as they are created). These same techniques can be used to maintain an ongoing thread of conversation. We will further explore this topic in the next section.

The last style of organization, which is seldom used but needed in some circumstances, is the creation of explicit substreams. These substreams are similar to directories because the user decides what goes into each one. Users create them by assigning a unique keyword[2] to each document in the substream (such as the keyword

[2]This is current done explicitly as described in Chapter 4 but could be implemented through a "stamping" or drag-and-drop mechanism.

`mydissertation`) and then creating a substream based on that keyword. This form of substreaming can be used to mirror an entire file system. For example, if we want to place the directory `/etc` in a lifestream we can tag all of its files with the keyword `/etc`, and so on recursively for the entire directory structure. A find on `"/etc"` would then retrieve the files in that directory. At the same time the files in `/etc` remain available to be found by other find operations.

5.2 Electronic Mail

Electronic mail is the most prevalent application used by computer users and fills a variety of roles beyond information transfer between individuals [WS96]. Sending and receiving electronic mail in Lifestreams is similar to what the user is already accustomed to: to send a document the user creates a new document and uses the transfer operation. Similarly, existing documents are easily forwarded to other users, and documents can be cloned and replied to. Incoming email is automatically appended to the stream.

While users normally handle "one-shot" communication on the main stream (e.g., a new message arrives and the user reads it and possibly responds), persistent substreams can be used to monitor an ongoing thread of conversation and switch back and forth between multiple conversational threads. One of our local users maintains an ongoing conversational thread with her external thesis reader by creating a substream with the search query `"to:suresh or from:suresh"` (`to` and `from` are tags as described in Chapter 3). This query uniquely captures all her mail to and from the reader. She often uses this substream as the basis for finding more specific messages from their conversation (such as `find semantics` or `find visit`).

All mail messages (incoming and outgoing) are intermixed with other documents in the main stream; however the user can easily create a mailbox by substreaming. In our prototype all transferred documents are automatically tagged as `email`, so the mailbox can be created through `find type:email`. By incrementally substreaming from the mailbox substream the user can create conventional mailbox "folders." These folders might contain all mail from a particular user (or group of people), all mail about a particular topic area, or the mail a user has not responded to. Because these folders are substreams they also act as filters and continue to collect new mail as it arrives.

The user can also "remove" email from a substream (or the main stream) by finding `"not type:email"`. This will create a substream containing all the documents that are not email. This also has the effect of filtering out incoming email[3]. When the user no longer needs email to be disabled, he can remove the incrementally created substream and return to the original substream.

[3] As the prototype is currently implemented the user can't filter out incoming email without also filtering out old email messages that belong to a substream. With the addition of time qualifiers on searches (e.g., `not :email after 3pm`) this shortcoming can be alleviated.

We have already mentioned how users can dial to the future and deposit documents that act as reminders. A user can also send mail that will arrive in the future if he "dials" to the future before sending a message. When the message is transferred it will appear in the future of the recipient's stream and (like a reminder) only be visible to the recipient when he dials to the future or when its creation time arrives. We use this ability to send mail to the future to post reminders to others about important meetings, department talks, etc.

Last, a user can summarize a substream of email with the default summarizer as seen in Figure 5.1. Here the user has summarized all the email from a particular user. For each email message in the substream the default summarizer strips off its header, displays the subject along with the creator of the message and then the first n characters of the message (here $n = 80$). While there is nothing computationally "clever" about the default summarizer, the overviews it generates can nevertheless be useful. This version of the summarizer only creates a text version of the summary. Future versions could work in more of a hypertext manner, allowing the user to click on a summary, which would open the document within Lifestreams.

5.3 Contact Management

There are a number of contact managers on the market that store electronic business cards, the date and time of contacts, and time spent on tasks for billing purposes.

Lifestreams supports two MIME document types that can be used for contact management: a business card and a phone call document. The business card (shown in Figure 5.2) is a document that contains the information you would expect: name, title, company, address, phone numbers and email address. The phone call record stores information about a particular phone call (as most contact managers do): time of call, whom was called, phone number, and notes about the call (as seen in Figure 5.3). Note that the user could also just type contact information into text documents.

To add a new business card or a phone call document, one clicks on the `New` button and then changes the type of the document to `businesscard` or `phonecall` (via the attribute dialog box as shown in Chapter 4). In the case of a businesscard, users may `xfer` cards to each other (rather than having to enter each card into their lifestream). For phone call records, the user often needs to refer to a business card to obtain a phone number. This can be achieved by substreaming on the name of the callee, selecting their phone number and creating a new phone call record, pasting in the information required. Since this requires a few "mouse clicks" we have automated much of the task of creating a phone call record with a personal agent. This agent is included with our standard Lifestreams system and is located in the "automated tasks" menu. When the user wants to make a call they choose "Make Phonecall" from the menu and the agent is spawned and displays the dialog box in Figure 5.4. While the agent was created by an advanced user, the agent code size is roughly 150 lines, only 20 of which are

```
DOCUMENT SUMMARY

evaluation chapter <Benjamin B. Bederson>

     Hi Eric, Good news. I read your chapter, and have no comments. I
     am happy with the new version. I think the info from your logging
     was really useful in supporting the results of QUIS, and I will
     insist on logging for future

Re: WIRED/evaluation <Benjamin B. Bederson>

     I got the new chapter, and will try and read it this week. I was
     actually pretty happy with the CyberTimes article. Considering how
     bad reporters usually do, I felt that he actually had some idea
     about why Pad++ and Lifestreams are interesting. And, I thought
     his criticisms were

Re: Thesis comments <Benjamin B. Bederson>

     I would be happy to come, but unfortunately, I have just been
     traveling (and missed too many classes) this semester. Go ahead
     and hold the defense whenever is convenient, and I'll plan on
     coming for a visit either over x-mas break or early during the
     Spring semester. Will you still be around?

Thesis comments <Benjamin B. Bederson>

     Eric, Well, I finally managed to make time to read your thesis. I
     apologize for the delay and hope my comments can still help you.
     It might be a good idea to show these comments to David - but I
     couldn't find his email

Re: Questionnaire <Benjamin B. Bederson>

     All sounds good. I don't think having short-term users will give
     you much useful data. My recent experience with testing Pad++ was
     that Pad++ novices didn't do that well, even after a specific
     training period. It seems that it really takes people a while to
     learn new systems - not too surprising...

dissertation <Benjamin B. Bederson>

     I, unfortunately, am behind. I read the first two chapters and was
     very happy with your organization, and motivation. I have some
     small comments, but I was going to wait and collect them. I'll try
     to read through the rest by early next week. We got the CHI papers
     out, but now I have an ONR proposal due on Monday...
 .
 .
 .
```

Figure 5.1: A Summary of Email.

Figure 5.2: A Businesscard Document.

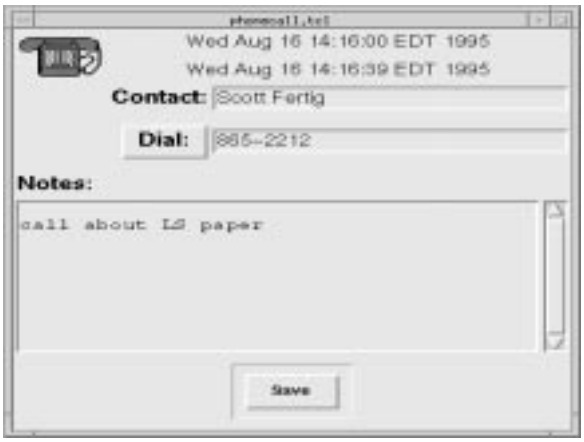


Figure 5.3: A Phonecall Document.

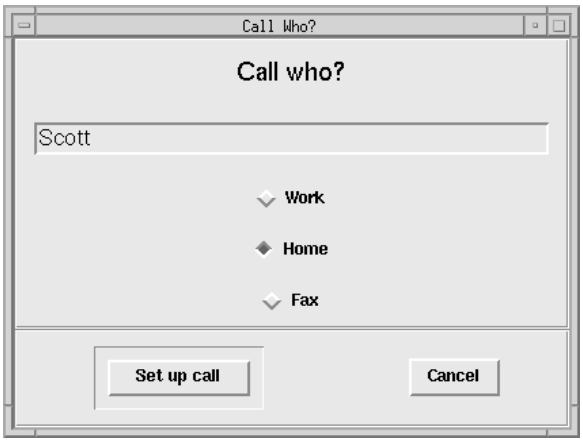


Figure 5.4: The Phone Call Personal Agent.

Lifestreams-specific code.

Once the dialog box appears, the user types in the name of the callee. The agent then searches the current stream for a business card with that name and, if found, creates a new document (with a MIME type of a phone-call record) and fills in the appropriate entries (Figure 5.3). This functionality is similar to the use of the personal assistant on the Newton platform. In the event that several records are found the user is given a choice. Note that there is an advantage to the chronological order in this case — as people update their business cards we can always get their most recent card by its position in the stream (the most recent appears closest to the head of the stream).

Substreaming can be useful in other ways in this context. For instance a user can browse though his "rolodex" by substreaming on `find type:businesscard`. Or he can refine his rolodex by incrementally substreaming to look at, for example, all contacts from `"find ibm"`. The user can do the same with his phone calls, looking at a list of all phone calls or selective phone calls.

The user can also summarize phone calls into a summary (shown in Figure 5.5). Phone call information could also be used to generate billing information for consultants (although we haven't yet developed a summarizer to do this). While a business card summarizer doesn't exist, an electronic rolodex would be a suitable form for the summary.

Lifestreams could also be extended to subsume the functionality of a time manager. Time managers generally track the billable hours a professional spends on one or more projects. In Lifestreams this is easily accomplished by creating a timecard that marks the starting and ending time of each task (these timecards are just thrown onto the stream as they are used). Then, before each billing period, the stream is summarized by the timecards, resulting in a detailed billing statement for each contract.

```
WHO             ON                              AT            ABOUT
-------------------------------------------------------------------------------
Ward Mullins    Thu Oct 31 11:50    EDT 1996  415 224-1912  Tcl/Java discussion
Scott Fertig    Wed Oct 30 12:05    EDT 1996  432-6433      Beth's Birthday
Susanne Hupfer  Mon Oct 28 10:06    EDT 1996  203-433-1211  Java Book/Class
Lorraine        Tue Sep 17  1:00    EDT 1996  488-1234      Crown Tower Deposit
Susanne Hupfer  Mon Sep 02 14:23    EDT 1996  203-433-1211  Cat
  .
  .
  .
```

Figure 5.5: A Summary of Phone Calls.

5.4 Managing Bookmarks

Many commercial tools have been created to help users manage web bookmarks. In our local research group we have found it difficult to keep track of our own web bookmarks and inconvenient to pass interesting bookmarks to one another.

In Lifestreams we have developed a system similar to "warm lists" [KM95], whereby a daemon watches the user's bookmark file, and each time a new bookmark is added the same bookmark is added to Lifestreams as a new "URL document." This URL document contains two pieces of information: the URL needed to connect to the remote site and the text of the page at the remote site. When a URL document is opened in Lifestreams the web browser comes to the foreground and displays the page by connecting to the URL.

Displaying a bookmarks list in Lifestreams is accomplished by substreaming on documents of type `URL`. In this way we can use Lifestreams to create a bookmark substream while at the same time making the data in the bookmarks readily available to any other searches we might make on our stream. This has an advantage over web browser bookmarks in that we can search over the text of each web page in our Lifestreams bookmarks (although this functionality has recently become available in commercial bookmark management applications). Moreover the bookmark substream itself can be incrementally searched for various types of bookmarks.

Passing URLs around to other users is also improved. Before Lifestreams, we usually traded URLs by copying an URL from a web browser to an email message, which the recipient would copy from email back to their own browser and add as a bookmark. With Lifestreams, passing URLs around is trivial. We `xfer` the URL document to another user's stream (a one-step process) and the URL is automatically included in their bookmarks substream.

We can recreate a bookmark "web page" by summarizing the bookmarks substream

(or any substream incrementally created from it) as shown in Figure 5.6. For each URL document in the substream the summary contains its title as a hyperlink (the user can click on the link and be connected to the page) as well as a description of the page that is created by parsing the content of the page's HTML and displaying the first 80 characters.

5.5 Calendar Applications

Lifestreams provides a natural data structure for the construction of many todo lists, calendar and scheduling applications. In the course of this dissertation we have only scratched the surface of such use, but our work suggests future directions.

Users of the current system have devised several ways of managing todo list items and calendaring,[4] some of which involve extensions to the system via summarizers. We begin with the simplest methods of managing todo lists. In all cases the users typically store one todo item per document. In the simplest case the user adds todo items to the stream on the date of its deadline. The user can then travel to the future and examine the list. If the user is using plain text files for todo items, then the word "todo" can be added to the text if the user wants to substream while in the future to retrieve only todo items (as opposed to scheduling information). When the deadline of the item arrives the user is alerted and can deal with the task immediately or clone it "back to the future" to put the task off (several users have requested a more automated way to "put off" todo items).

The user can also summarize todo lists. While the default summarizer would do, the author and another user worked together to create a summarizer for the future that represents the future of the substream as a weekly or daily planner. The summarizer decides which of the two is appropriate by the amount of time the user has traveled into the future — if the user travels one day into the future then a day planner is displayed (Figure 5.7), otherwise a week planner is displayed (Figure 5.8). Future work could also include longer-term views such as weeks and months.

Another clever (if less elegant) method is to tag todo documents as they are created or arrive from other users via xfer (or external email). For instance, one user tags todo list documents with the explicit keyword `todo`. When the tasks are completed she then tags then with the explicit keyword `done`. When she wants to examine her todo list she creates a substream by `"find todo and not done"`. While this method has some of the flavor of a location-based system, it nevertheless maintains advantages over those systems (e.g., these documents remain accessible by other searches, etc.).

Andrew Larratt-Smith, a Yale undergraduate, developed a complete calendar interface (not as a summarizer but as an alternative interface) to Lifestreams for his senior project [LS96]. The calendar is fully integrated into the X Windows interface and the

[4] These methods seem to be indicative of cognitive styles of using the system. We leave such analysis for others to explore.

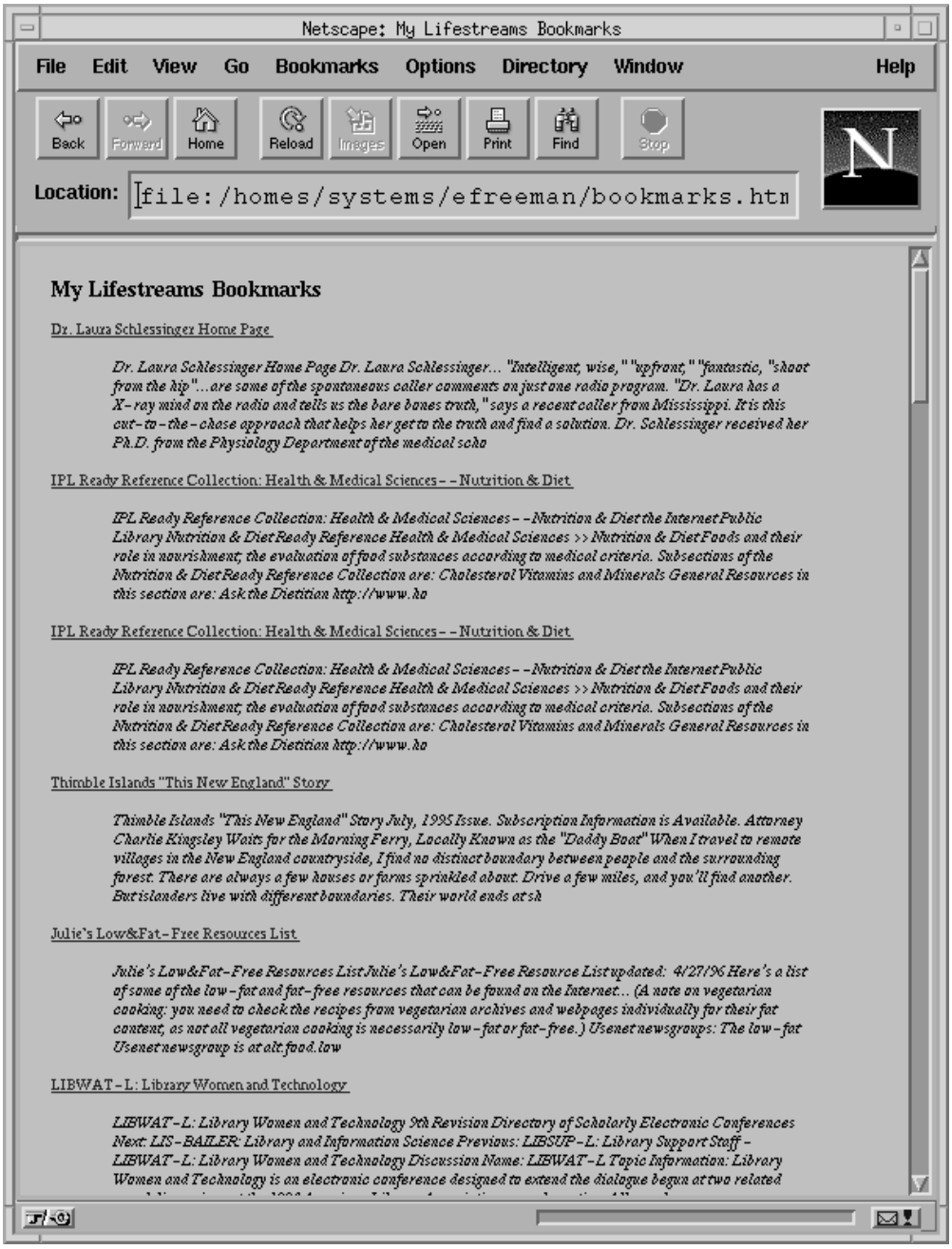


Figure 5.6: A Summary of Bookmarks.

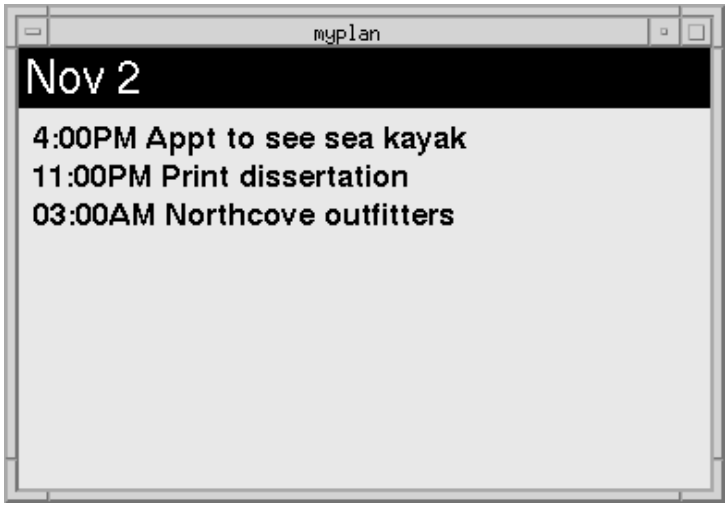


Figure 5.7: Day at a Glance Summarizer.

user can switch between the two using the `Display` menu. His interface is shown in Figure 5.9.

This interface is particularly useful when viewing the future portion of the stream allowing appointments, reminders and scheduling items on the stream to be displayed in a format that is familiar to users. Note that the interface, despite its departure from the stream view, still contains the Lifestreams "dashboard," that is, all the Lifestreams operations. The interface also provides some novel features. The user can select a day, click the `new` button , and create a document in the future by specifying a time. The user can also select a day in the past or future, double click and he is returned to the stream view focused on the appropriate time within the stream.

We have already discussed in the previous section on electronic mail how communication and future documents can be used to allow reminding of others. These techniques can be combined with personal and document agents to implement a number of groupware applications on top of Lifestreams. As a demonstration we provide one application here: a meeting maker.

To use the meeting maker the user first selects `Schedule a meeting` from the automated tasks menu. This spawns a personal agent that prompts the user for the names of the attendees (Figure 5.10). After entering the names the user is then provided with an overview of the week (Figure 5.11[5]) with conflicting meeting slots already blocked out. These dates are determined by examining the streams of all the attendees.

Once the user selects a meeting time a document is sent to each potential attendee. This document includes a document agent. When the user reads the message, the

[5]The interface code was originally written by Susanne Hupfer as a Turingware meeting maker [Hup96] and modified to work for Lifestreams.

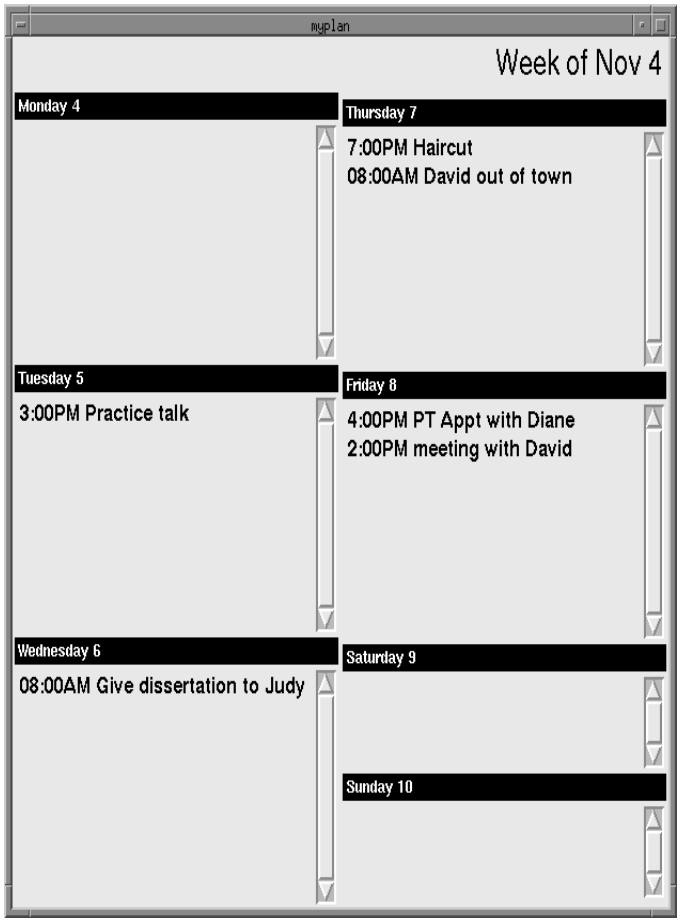


Figure 5.8: Week at a Glance Summarizer.

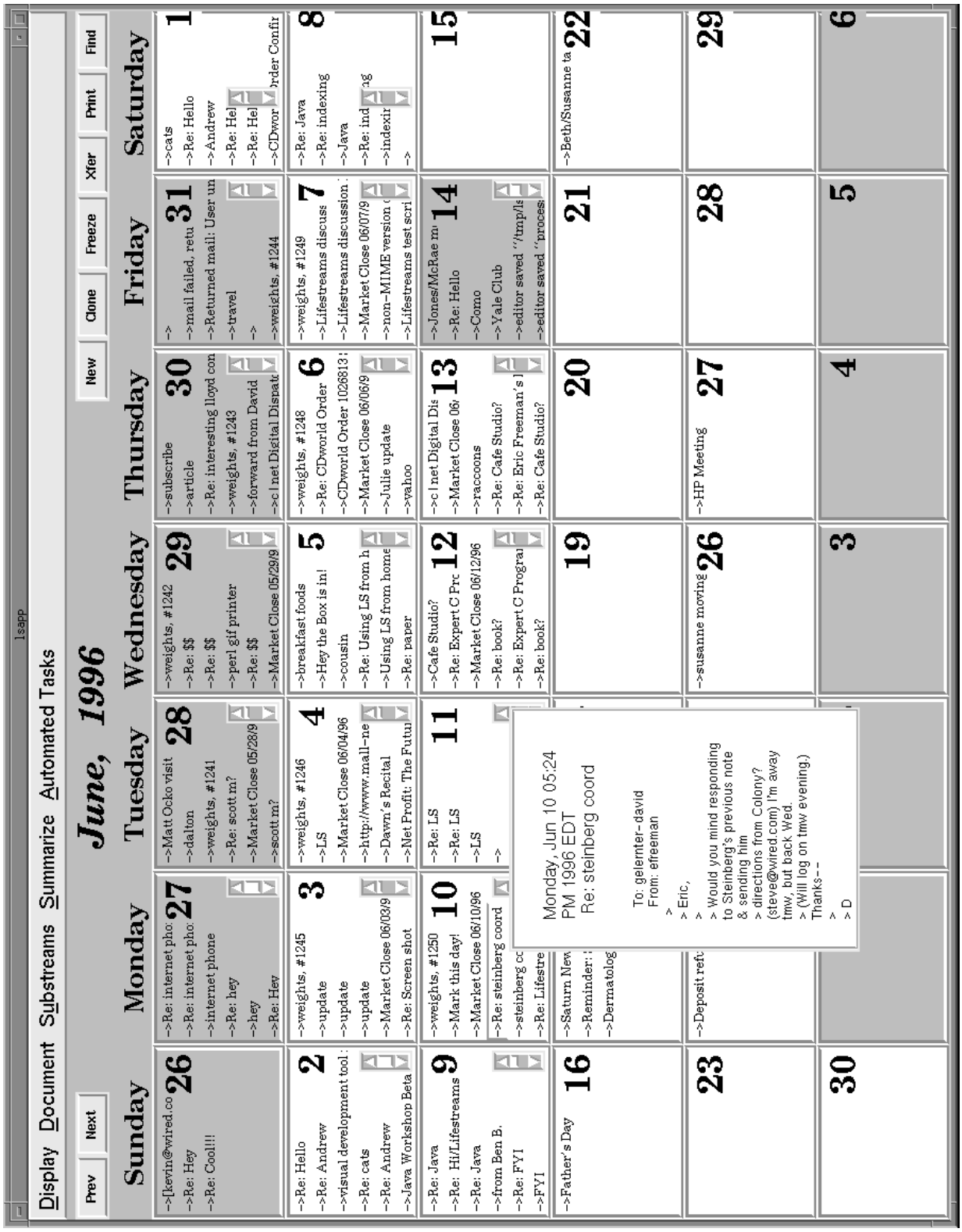


Figure 5.9: The X Windows Calendar Interface.

Figure 5.10: Meeting Maker Requesting Recipients.

Figure 5.11: Meeting Maker Scheduler.

agent asks the user if he can attend or not (Figure 5.12) and a message is returned to the initiator's stream.

The personal agent on the initiator's stream waits in the background until all responses are collected. If all users okay the meeting then a future document is placed in each attendee's stream. Otherwise, a document is placed on the initiator's stream to indicate the scheduling failed.

While there are many obvious enhancements we could make (automated retries of failed meetings, finer granularity of meeting times, etc.), we offer the meeting maker as an example of a straightforward extension of the system that automates a process for the users and makes use of the time-ordered nature of the stream.

5.6 Personal Finances

We have just begun to explore using Lifestreams to manage personal finances. While we have experimented with using Lifestreams to track stock portfolios, providing other financial services within Lifestreams would require us to simulate electronic banking or to work closely with an existing service — both of which are beyond our current resources. We will, however, suggest the ways Lifestreams can be used in personal

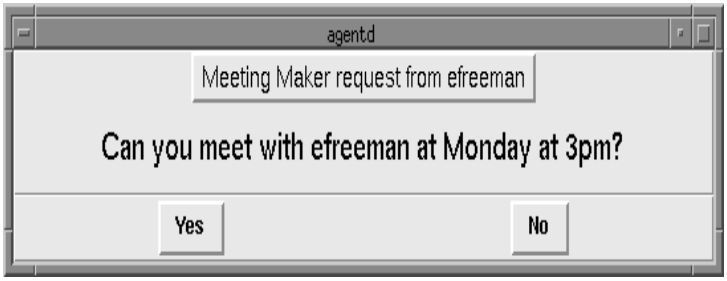


Figure 5.12: Meeting Maker Document Agent.

finance.

Millions of typical users already track their checking accounts, savings, investments, and budgets on their computers with applications such as Quicken. The types of records and documents used in applications such as Quicken — electronic checks, deposits, securities transactions, reports — can be conveniently stored and generated by Lifestreams. A user could easily migrate his checking account to Lifestreams so that each check written creates a record on his stream. Some of these checks would be electronic checks sent to companies with an online presence; others transcribed from written checks (just as many people already do with Quicken). A substream would recall all electronic transactions, all checks, or all bills. The user could then employ a summarizer to help balance his checkbook. At year's end summarizers could be used to help a user prepare his taxes (which could then be shipped electronically to the IRS).

Lifestreams could also help with budgeting, tracking expenditures, etc. Of course, many of these capabilities are already available in products like Quicken; but it is worth pointing out that Lifestreams contains *everything* a person deals with in his electronic life. Rather than storing financial records in a monolithic file, each could be stored separately in Lifestreams and made use of by many possible applications (e.g., summarizers, agents, etc.)

For managing portfolios many services that forward daily closing values of securities via electronic mail already exist. The body of an email message from one such service is shown below (from Net Profit, Inc.):

```
   DJIA: DOWN 5.55 to 5674.28   NYSE: DOWN 1.06 to 352.81
 S&P500: DOWN 2.26 to 660.23   AMEX: DOWN 0.38 to 548.59
RUS2000: UP 0.33 to 324.74 NASDAQ: DOWN 4.39 to 1120.53

             Shares    Today's   Today's         Total      Total
Symbol        Owned      Close    Change         Value     Change
-------- ---------- ---------- --------- ------------- ----------
```

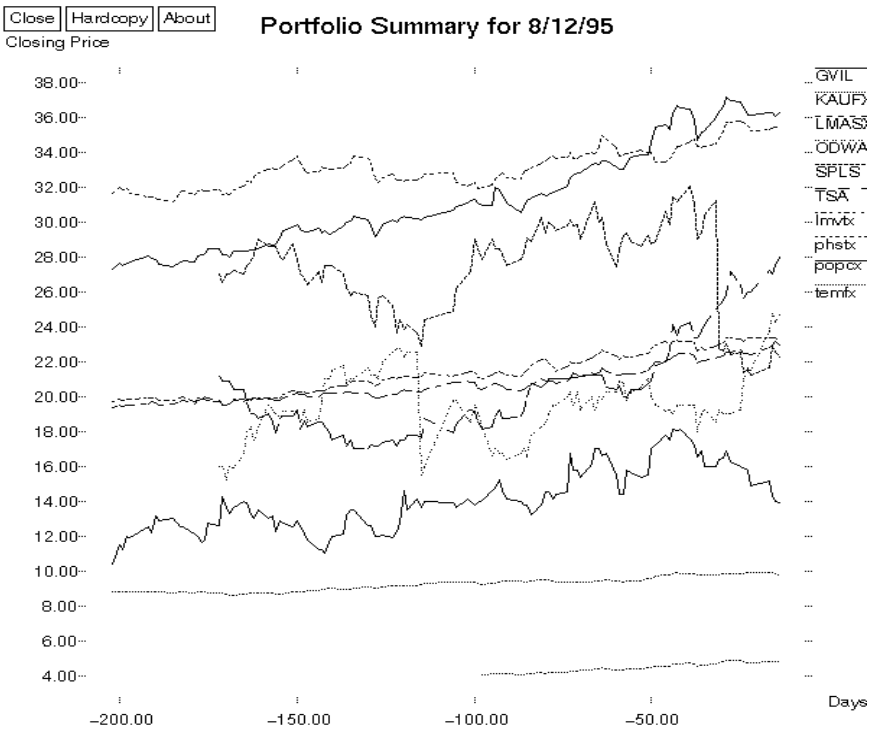


Figure 5.13: A Summary of Stock Performance.

GVIL	350.000	7.000	-0.125	2,450.00	-43.75
ODWA	400.000	18.750	-0.500	7,500.00	-200.00
SPLS	112.500	17.250	-0.125	1,940.62	-14.06
TSA	100.000	19.000	0.125	1,900.00	12.50
TSSW	600.000	3.000	0.000	1,800.00	0.00
				15,590.62	-245.31

Users can easily access all such records in their lifestreams via substreaming (in this case `"find from:netprofit"` uniquely captures the stream). The user can then browse over each day's records. To view the historical performance over time the user can make use of `summarize` as shown in Figure 5.13. This summarizer collects the history of the portfolio over time and provides a chart showing performance.

5.7 Summary

The domain-specific examples we have examined in this section illustrate the unifying effect that Lifestreams has over data and applications. In Lifestreams the same opera-

tion that creates a generic document (`new`) also creates an email message, a reminder, a business card and (indirectly) a web bookmark. Likewise, the same operation that creates a "working directory" also creates an electronic mailbox, a "hotlist", and a contact list. Browsing a "directory" in search of a file, a mail box in search of a mail message, a contact list in search of a contact, and an appointment schedule in search of an appointment are all the same operation. Moreover, if you know how to use `find` to locate a document, you also know how to set up an "agent" to filter your email (via a persistent substream). In every case, `summarize` gives you a synopsis — a summary of files, a summary of bookmarks, a list of contacts, a summary of mail, or your schedule at a glance.

Chapter 6

Evaluation

Thus far we have motivated Lifestreams, presented a model for the system, described an instantiation of the model (the research prototype), and looked at the prototype in the context of common user tasks. In this chapter we evaluate the research prototype; our goal in doing so is to support the thesis, to support or disprove any intuition we have about the model, to shed light on the strengths and weaknesses of the current model and system, and to drive the development of next generation Lifestreams systems.

6.1 Scope and Methodology

The creation of experimental computer systems is an incremental pursuit. While the goal is to provide novel tools that improve the utility of the computer, the resulting system is often the result of incrementally building prototypes, each based on what was learned in the previous generation. Early computer systems were designed and used by technical people. Today's systems have to be functional over a wide range of users (or at least Lifestreams aims to be so). Shneiderman has claimed that the programmer's intuition is no longer good enough, systems have to be validated through prototype, usability and acceptance testing [Shn92].

In this dissertation we describe the first steps toward this evaluation. Our study of Lifestreams is based on over 24 months of user feedback from six users (seven, including the author). In the process three generations of user interfaces were incrementally built and numerous changes were made to the semantics of the underlying system.

We are interested in evaluating two key aspects of the system: usability and utility. Usability addresses the question "was the user interface (and in our case also the "metaphor") acceptable to the user?" This acceptability can be evaluated by measuring several aspects of the interface [Nie93]: How easy is it to learn? How easy is it to use? Is it subjectively pleasing to the user? Utility is a measure of the overall efficiency and effectiveness of the system. While usability is typically a subjective measure and utility a quantitative one, we will see that the two can be used together to obtain more

meaningful analysis than using either measure alone.

In general, our efforts can be described as a small-scale field study (involving no control) with subjects who have used the system extensively. To this study we apply the best available techniques for evaluating user satisfaction, and the usability and utility of the system. To measure user satisfaction we rely on the *Questionnaire for User Satisfaction* (QUIS), a standardized usability testing tool available for licensed use from the University of Maryland. The QUIS tool was developed by Shneiderman and Norman [Shn92] at the University of Maryland and attempts to capture the overall subjective satisfaction with an electronic system. Subjective satisfaction is important in that it is a crucial measure of an electronic system [HN93] — although a system may be evaluated favorably on performance measures, it may not be used because of user dissatisfaction. Using a standardized tool avoids many of the problems of self-developed questionnaires (question bias, variability, reliability). QUIS is currently the most thoroughly developed testing tool available, having been widely used in industry and academia, and experimentally shown to have high reliability and low variability when measuring subjective satisfaction [CDN88]. Beyond measuring user satisfaction, QUIS can also be used to diagnose the strengths and weaknesses of a system and to assess potential areas for system improvement (as we shall see) [Nor90].

Our second tool for evaluation is the use of program instrumentation to collect data on the utility of the Lifestreams system. We do so using conventional, well-established techniques for monitoring system usage [Per96]: keeping a time-stamped sequence of user commands along with any important system variables (such as the keywords used in a search query) in a log file. The log file can then be used to determine which parts of the system were used and not used (unused features often turn out to be hard to use or not useful), to examine differing usage patterns between users and to provide data on system speed and accuracy.

We supplement both of these measures with a user survey consisting of specific questions about various aspects of the Lifestreams. While data from QUIS and instrumentation logs can provide specific measures, in many cases they don't provide enough specific information to determine why users responded as they did or how a utility measure might have affected the user's experience with the prototype. For example, we can determine the success rate of substreaming from the instrumentation data but we have no way of knowing how that rate affects the user. Without the users feedback we might find it hard to determine if a specific success rate (say 90%) was either acceptable or frustrating to the user. We also use this survey to gain additional acceptability data from the user. For example, we would like to know not only the overall satisfaction (on which QUIS provides data) but also the user's reaction to specific aspects of Lifestreams, such as chronology and on-demand organization.

Finally, we also make limited use of the feedback from one additional source: "observers" of the system. Over the course of the prototype development observers read papers about the system (that included pictures of the interface), viewed an ACM video that includes a demonstration of Lifestreams, and saw live demonstrations. Life-

streams was first demonstrated to roughly 20 individuals at MIT in the fall of 1995. Since then over two dozen people have viewed in-depth demonstrations of Lifestreams at Yale mostly in a one-on-one setting. These people have come from many different disciplines: writers, entrepreneurs and computer scientists.

In summary, we will use data from three specific tools to evaluate Lifestreams: QUIS, prototype instrumentation and a user response survey. Each tool provides a piece of the complete evaluation; QUIS provides a reliable and repeatable measure of subjective user satisfaction but offers little information about why the user was satisfied (or not). The quantitative data from instrumentation provides us with a sense of how subjects actually used Lifestreams and data on the success rate of certain operations (substreaming) but does not tell us if these rates were acceptable to users. Last, the user response survey brings the two measures together by gathering more specific information about the subject's reactions to the features, operations and functionality of Lifestreams.

We now lay out the specifics of our study by describing our subjects and the procedures used to introduce them to the system and collect data. We then begin our evaluation by using QUIS and the user survey to examine the users' subjective reactions along several variables: the overall system, the "metaphor," its learning curve and the interface. Next, we examine the users' subjective reactions to the system's performance and capabilities (again through QUIS) and then move on to analyze the instrumentation data, bringing in responses from the survey as needed.

Before moving on we wish to point out that, in general, the small subject size of our study prohibits us from showing any type of "statistical significance" but we believe our methods provide more general utility than a larger study of more narrow scope. Nevertheless, our work is limited and further analysis work remains.

6.2 Subjects

A total of six users participated in our study (seven including the author). This group consisted of two Yale undergraduate students, two Yale computer science graduate students, and two Yale staff members. The staff member's jobs were of a clerical/administrative nature. Two of the group members were female and four were male (mean age = 34.3 years). Each member of the group used Lifestreams for a minimum of one month (mean time = 8.2 months). Three of the users reported that they used the system ten or more hours a week, two of the users reported they used the system between four and ten hours a week and one user between one and four hours a week. The two graduate students had advanced computer skills, the two undergraduates had intermediate skills and the two staff members were novice users.

6.3 Procedure

6.3.1 Introduction and Training

Subjects were provided with the Lifestreams X Windows client running on their workstation. For each user we "bootstrapped" their lifestream with old mail spools, the files in their directories and their web bookmarks (each subject was asked what specific mail messages, files and bookmarks they wanted to put on their lifestream). Each client included personal agents (reminder, freeze all, mark all read, phone call) and summary types (default, bookmarks, LaTeX, future). Subjects were shown the basics of using the system (five to ten minutes of demonstration) and then left to explore it on their own (although subjects were allowed to ask for help at any time). No subjects (including the two staff members) were given "out-of-band" time to use and explore Lifestreams; rather they were asked to try to incorporate it into their daily activities.

6.3.2 Evaluation

In October of 1996 the subjects were given a questionnaire containing two parts: a set of questions from the QUIS usability testing tool and the Lifestreams-specific survey.

The QUIS questionnaire contains a demographic set of questions (age, sex, etc.) and a set of questions that measure overall system satisfaction. Each question measures the users' overall satisfaction on a 9-point scale (1 to 9)[1] that is anchored at the endpoints with "semantic differentials" (difficult versus easy, terrible versus wonderful, never versus always). One is always the bad score and nine the good score. An example question is

```
6.1   Learning to operate the system was
              difficult                        easy
                      1  2  3  4  5  6  7  8  9       NA
```

Questions are grouped into categories, such as overall reaction, learning, and system performance. The standard questionnaire contains 32 questions and administrators are encouraged to tailor the questions to their needs (by possibly removing questions that are not relevant). Our questionnaire contains 22 questions from four categories.

Attached to the questionnaire the users were given a survey with 24 questions to be answered in essay form. Our intention was to gather more specific information about their reactions to various aspects of Lifestreams. The questionnaire and survey can be found in Appendix B.

For a thirty day period we also collected data from four users by instrumenting the Lifestreams client to log system use. This data was collected near the end of

[1] A 1-10 scale introduces a bias when the questionnaire is administered electronically as typing "10" requires two keystrokes. Likewise, zero is not used because it is used to encode N/A in the electronic version.

Overall User Reaction to System	
Question	Average Score
(terrible/wonderful)	8.5
(frustrating/satisfying)	8.3
(dull/stimulating)	8
(difficult/easy)	8.5
(inadequate/powerful)	7.5
(rigid/flexible)	7.8

Table 6.1: Subjects' overall reactions to Lifestreams.

the dissertation work (between September and October 1996) when the system was mature and most reliability problems had been addressed. The subject group for these tests consisted of the two clerical workers and two graduate students[2]. The two undergraduate students were no longer available for testing. Each operation the subjects performed was recorded to a log file. Logging was performed in a noninvasive way and was invisible to the user. Users were told that their operations were being logged but assured that they were not being "tested."

6.4 Overall Subjective Reaction

QUIS contains an initial set of six questions that attempt to measure the overall subjective reaction of users to a computer system. When answering these questions (Table 6.1) subjects reported a high overall subjective reaction to Lifestreams (the mean score was 8.1 out of 9 over all six questions). Average scores were very positive on the ratings of easy, satisfying and stimulating, while only slightly lower on power and flexibility (we will discuss user comments on power and flexibility in more detail later in this chapter).

These findings are consistent with the reactions reported by users on the survey (as characterized by this response):

> *The concept (of Lifestreams) appealed to me immediately on two levels. First, because I know myself that I naturally order and recall events in my life according to time cues, that "memories" become less important to my daily activities the further in the past they recede (yet retain punch and applicability at discrete moments when recalled because of similarity to*

[2]One of the two graduate students left the study before the logging period. The author's logging data was used in its place.

> *current events), and that I find it so incredibly annoying not to be able to recall something that might be applicable because the "index" to that memory has been lost, or that a relevant document is no longer available because it has been thrown away (just weeks before to remove "clutter" or save space).*

These findings are also consistent with the reactions of observers (although we can only "measure" the satisfaction of actual users) during the course of developing the prototype. Observers typically had a strong reaction when viewing an in-depth demonstration. It was also common for observers to send mail of the following nature (in this case after seeing a demonstration the day before):

> *...the time I now spend on this system of mine has really changed. I hate hunting through this "Tree". It is cumbersome at best and annoying at least. I have seen a better way. I didn't realize how much time I spend searching for documents.*

Even without a demonstration most people reacted enthusiastically to the concept after reading a description or viewing a video. The following email comment from an observer characterizes these responses:

> *I have so much stuff coming in my "InBox" daily, whether it's incoming e-mail, snailmail, phone messages, articles, or what-have-you; that there's not really time to organize it all. Rather, as you quite convincingly point out, I'd rather just STORE it all (since storage is cheap!) and access only what I want when I want to access it.*

Beyond the QUIS questions we were also interested in how people reacted to the novel aspects of Lifestreams — did chronology make sense as method of storing data? did they understand the "stream metaphor?" how did the system compare to their previous environments? — and we asked many questions in the survey to determine their reactions.

6.4.1 Reaction to chronology

Most users and observers immediately "got" the idea of the chronological storage system. We found that at least one user was comfortable with the chronology-based storage because he commented that the "way I file everything in Unix is on the top level." This combined with listing a directory by time (e.g., `ls -t`) gave him something similar to Lifestreams storage.

Users also easily understood the "working" versus "archival" aspects of the stream:

> *The majority of my job is made up of constant little emergency items. This depends not only on current messages, but issues that may have come up as far back as when we came to this building (building repairs, furniture, etc.) Up until now I've only had paper to depend on.*

Other comments from users about the time-based storage were less insightful but encouraging. One user simply responded "so far [chronology] has been great for me." Only one user found the concept of time-based storage strange at first but responded on the questionnaire that it was "hard to believe [chronology] is as easy to get used to as it is."

Asked about the use of chronology as a storage scheme, users commented that "chronology should definitely be the default." Another user commented that she would like to be able to order documents by creator and type, however she would not "prefer it over chronology but rather as an alternative view" (underline in original response). Another user suggested that Lifestreams have buttons that would let her "switch schemes as you would a stove (broil, bake, preheat)."

Users also appeared to easily understand the use of the future portion of the stream. One of the clerical workers was able to extrapolate and suggest its use within her workgroup:

> *...my area, the Business Office, ... would greatly appreciate the "future" email messages to themselves and others for countless deadlines that they now must use calendars and daytimers for.*

6.4.2 Understanding the "metaphor"

Five of the six subjects said that the "metaphor" made sense to them (the sixth wrote that he didn't understand the question). Subjects were also asked if any aspect of Lifestreams was confusing to them. Four subjects answered no. The fifth subject reported that the only confusing thing ("at first glance") was the lack of directories. The remaining subject reported that he found the distinction between frozen and unfrozen documents confusing. This comment deserves future study as it may be that novice users are not used to thinking of computer data as read-only or writable.

6.4.3 How does it compare to other systems/metaphors?

When asked in the survey to compare Lifestreams to their previous "computing environments" users had a variety of comments:

> *Easy to use and intuitive*
>
> *Easy to get (an) overview feel of information*
>
> *The structure is dynamic and fluid — the data, not just the filename, is on the screen*
>
> *Easy to find documents of different types*
>
> *Searching! It's great to be able to search over everything in my stream.*

Question	Average Score
Learning to operate the system (difficult/easy)	8.8
Getting started (difficult/easy)	8.8
Learning advanced features (difficult/easy)	8.2
Time to learn to use the system (slow/fast)	9

Table 6.2: Subjects' reactions to learning in Lifestreams.

In a related question, subjects were asked how Lifestreams might have changed the way they think about using a computer or about managing information. Responses included "a different and easier way to file", "it made me think of information in terms of time rather than locations, as a flow rather than distinct slots", "files chronologically close in time are more likely to be related; easy to gain a historical perspective on any topic", "absolutely, (I) think, work with, and find my documents (i.e. content) without regard to creating application or storage location", "I used to be very concerned with storing/managing/trying to remember."

6.5 Learning

Learning received the highest user satisfaction rating of all the QUIS questionnaire categories — with a mean score of 8.7 out of 9 over all four questions (Table 6.2). Subjects responded with an average score of 8.8, 8.8 and 9 respectively on questions of learning to operate the system (difficult/easy), getting started (slow/fast) and the amount of time it takes to learn the system (slow/fast). The written response of one user is a good example of a user's subjective reaction to the Lifestreams' learning:

> *The time at which I started using Lifestreams was at the beginning of the semester, my busiest time ... All this considered, I was still bowled over by all of the ways it could, and did, make my job easier in a very short period of time.* (underline from original response)

The QUIS question on learning advanced features received the lowest score in the group (an 8.2). This is consistent with questions that were asked by users over the course of the study. Most of these involved how to clone or transfer future documents or the specifics of how summarize worked. Many of these problems could probably have been avoided if we'd set aside time for a training session rather than setting the user up, explaining the basics and letting them explore on their own.

Question	Average Score
Were the screen layouts helpful? (never/always)	8.3
Amount of info that can be displayed on screen? (inadequate/adequate)	7.2
Arrangement of information on screen? (illogical/logical)	8.3

Table 6.3: Subjects' reactions to screen layout and design.

6.6 Interface Layout and Design

All six subjects responded positively in the survey that they quickly understood the receding stream interface. In QUIS the subjects responded with an mean score of 8.3 on the usefulness of the screen layout and the arrangement of information on the screen. On the survey subjects responded that the stream was an "obvious" aspect of the system.

In QUIS subjects responded with a slightly lower score on the interface's information density and this was reflected by some responses in the survey. One student commented that the information density of the display was "low" and that a way of displaying more of each file's contents would improve the interface. However, another user thought that the "concept behind the screen layouts ... were excellent and made great use of the screen." We tend to agree with the first student and believe the data density of the interface can be improved.

Initially novice users did have one problem that was corrected before the testing period — while the window manager capabilities of our client were useful for advanced users they proved confusing for novice users. These users had trouble understanding the model of external applications handling the editing of their documents that would then be written back to the Lifestreams system. We believe this is because they did not have a good mental model of the process. Novice users also disliked the multiple mouse clicks needed to send mail messages or reply to existing messages (e.g., New-Edit-Save-Xfer or Clone-Edit-Save-Xfer). Rather than teaching these users this "mental model" we chose to make the system easier to use by supplying an editing window for text documents that opens within the Lifestreams client rather than within the window system (documents with other content-types still opened within helper applications). Since the editor stays anchored in the client window, the novice user does not have the problem of "losing" an editor window. This editing window is annotated with common operations such as `Reply`. The effect of clicking on `Reply` is that the client automates the task of cloning the document and transferring it. Novice users found these changes satisfactory and advanced users liked the convenience of the mail-related automation. In any case, both sets of users still had the underlying hypertools model for media types other than text and this seemed to create a happy medium.

6.7 System Capabilities and Performance

Nine of the QUIS questions addressed the capabilities and performance of the prototype. In addition we logged and recorded twenty variables of the system's use and performance. Looking at the responses to the QUIS questions in Figure 6.4, we see a range of answers from 6.6 to 8.6. High scores were given to the question about the ability of novice users to accomplish tasks (one of the main goals of this work) and to the dependability of the system operations (8.3). Slightly lower scores were given to the system's speed (7.8), the response time of operations (7.3) and the reliability of the system (7.4). While these are still highly positive numbers, it is important to point out that two of the prototype users (the undergraduates) used the system before the summer of 1996, before a number of performance improvements and bug fixes were made. If we only consider the scores of the other four users the average scores over these questions (speed, response time, reliability) are 9, 9 and 7 respectively. So while we improved response time and system speed, we still had reliability problems. We believe most of these problems are with the Solaris version of Lifestreams. When examining the data of the only subject who used the Solaris version we find that he gave the question a score of 5. Given that all other users (including the developer) were on a SunOS version of the system it is possible that the Solaris system received less attention when it came to removing bugs and glitches caused by Solaris. This problem shows up again in the QUIS question pertaining to system failures, which received the lowest score (6.6) in the entire questionnaire. Here the Solaris user reported a score of 4. The average score of the remaining users was 8.5.

Finally the questionnaire asks users about suitability of the prototype for experienced users. When asked if the prototype addressed the needs of both novice and experienced users the subjects responded with a mean score of 7.6. Given that the previous question about novice use received a high score we will attribute most of the deficiencies here to the needs of experienced users. When asked if experts could use features or shortcuts the subjects responded with a score of 7. On this point several users responded with written comments:

> *[A] command key would be helpful in avoiding redundant steps, e.g., Ctrl-N for new...*
>
> *A few more shortcuts could be added using command keys.*

So while the needs of novice users were accounted for, improvements can be made to address the needs of advanced users.

6.7.1 Overview of System Use

During the logging period a total of 5558 Lifestreams operations were performed by four users. In Table 6.5 we present each operation along with the number of times it

Question	Average Score
System speed (too slow/fast enough)	7.8
Response time for most operations (too slow/fast enough)	7.3
Rate information is displayed (too slow/fast enough)	8.3
How reliable is the system? (unreliable/reliable)	7.4
Operations are (undependable/dependable)	8.3
System failures occur (frequently/seldom)	6.6
Novices can accomplish tasks (with difficulty/easily)	8.6
Are the needs of both experienced and inexperienced users taken into consideration? (never/always)	7.6
Experts can use features/shortcuts (with difficulty/easily)	7

Table 6.4: Subjects' reactions to system capabilities.

was used and the percentage of its use. These numbers are provided for each user and for the group as a whole; here we include not only the primitive operations, but also document reads[3], "time travel" operations (going to the future, past or returning to the present), substream operations (accessing existing substreams, removing substreams), and also freeze and print operations.

As one might expect, reading/editing was the most common operation accounting for 40% of all operations (because writes were not logged there is no way to distinguish between read and edit operations). Document creation was the next most popular operation occurring over 21% of the time. Of the 21%, `new` accounted for nearly 13% and `clone` for 8%. The `transfer` operation occurred nearly 20% of the time. This was also to be expected given the amount of time the average users spend handling email — with 1084 transfers and 1159 document creations, many of the documents created were eventually sent to another user (although "incoming" documents are often forwarded through transfer without a corresponding creation operation). Operations used to manage substreams accounted for 7.8% of the user's operations — these operations occurred when subjects used the substream menus to recall an existing substream, remove a substream, or return to the main stream from a substream. Time travel operations accounted for nearly 5% of the user operations; the find operation occurred slightly less often at 3.7%. When we consider the substream and find operations together we obtain a measure of how often users "managed" information (by relying on organizational structures they had already created or in creating new structures) they account for 11.5% of the total operations. Summarize operations made up 1.4% of the total. Of the summaries performed 57% were future summaries, 33% were

[3] A count was not logged for the write operation.

	G_1	G_2	C_1	C_2	Total
Read/Edit	757 (39.8%)	980 (37.4%)	358 (49.5%)	156 (50.5%)	2251 (40.5%)
Xfer	367 (19.3%)	561 (21.4%)	95 (13.1%)	61 (19.7%)	1084 (19.5%)
New	220 (11.6%)	338 (12.9%)	104 (14.4%)	44 (14.2%)	706 (12.7%)
Clone	178 (9.3%)	244 (9.3%)	19 (2.6%)	12 (3.9%)	453 (8.2%)
Substreaming	238 (12.5%)	285 (10.9%)	95 (12.9%)	22 (7.2%)	638 (11.5%)
Time	85 (4.5%)	146 (5.6%)	32 (4.4%)	3 (1.0%)	266 (4.8%)
Summarize	32 (1.7%)	31 (1.2%)	7 (1.0%)	10 (3.2%)	80 (1.4%)
Misc	27 (1.5%)	37 (1.4%)	15 (2.1%)	1 (0.3%)	80 (1.4%)
Total Ops	1904	2622	723	309	5558

Table 6.5: Use of Lifestreams operations.

LaTeX summaries and 10% were bookmark summaries. The use of summarize is a little lower, but not really surprising given the small number of useful summarizers we have developed (and users did request new types of summarizers to be added to the system). Freeze and print each made up less than a percent of the total operations.

Overall, the subjects' use patterns were fairly uniform. Some notable exceptions are that the clerical users tended to use `clone` a little less than the graduate students while using the `new` operation slightly more often. While the reasons require further study, graduate students may reuse and edit information more often than clerical workers who handle more short term action items.

6.7.2 Use of Substreams

In Chapter 5 we discussed the various ways that substreams organize information. We now look at the data collected from use of substreams. During the logging period 207 find operations were performed. Of the 207 operations, 150 (72%) of them were keyword searches, while the remaining 57 (28%) where boolean queries. We further analyzed these searches to determine how many actual queries were performed; that is, the user may have used multiple find operations in an incremental fashion to satisfy one query. We also determined whether or not a query was successful by the following heuristic: after a find, if the user performed an operation on a document in the substream or on the substream itself (before removing the substream or returning to the main stream) then the query was considered a success. The valid operations included reading the document, cloning it, printing it, transferring it or summarizing it (or the substream). This heuristic may misclassify in at least two ways (both favorable and unfavorable). In the first case, a find may be counted as a failure when in fact the subject used the glance view capability of Lifestreams to obtain the information he needed. In the

	G_1	G_2	C_1	C_2	Mean
Number	63	95	23	6	46.75
Query Size	1.3	1.23	1.5	1.7	1.43
Query Depth	1.11	1.07	1.09	1.0	1.07
Success Rate	90.5%	94.7%	78.2%	83.3%	86.7%

Table 6.6: Query use over testing period.

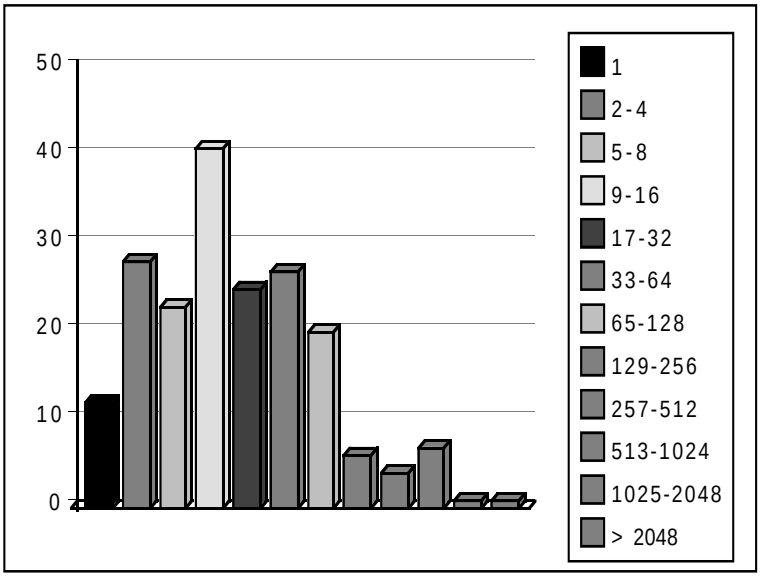


Figure 6.1: Size of substreams returned from FINDs.

second case the user may have opened a document within the substream that turned out to be the incorrect document. One other scenario is also not accounted for — a user may use multiple incremental finds in an attempt to locate information and then return to the main stream and begin again using a slightly different scheme for finding the same information. This can cause an inaccurate measure of the number of finds required to satisfy a query. For this last type of error we manually scanned the logs, identified obvious cases and then asked the user for confirmation. As an example one user performed a find on "book AND order" and then returned to his main stream and performed another find on "Dover" (a publisher). The user confirmed that these two finds were performed to locate the same information. While the heuristic would have counted this as two searches (one failing and one succeeding) we adjusted the data to show that a search consisting of two queries was performed and successful.

Our analysis also accounts for incremental substreaming — if, instead of performing an operation on a document in the substream, the user incrementally initiated another find, the depth of the query was incremented and the result considered a success or failure based on the result of the second find (and so on). We found that 20 (9.7%) of the find operations were performed incrementally. Of the 187 (207 – 20) queries performed, the average query depth was 1.07. 170 queries were counted as successes resulting in a 90.9% success rate.

In Table 6.6 we present the data for all users. The number of substreams varied among users from 6 to 95; this is partly a reflection of the amount of time the users spend at the computer. The two graduate students (G_1 and G_2) typically spend their entire work day at the computer, while the two clerical workers spend only a fraction of their time in front of the computer (especially in the case of C_1, who only works 20 hours a week). Nevertheless, the results for query size (number of search terms in a query), query depth (number of finds required to locate a piece of information) and success rate are fairly uniform. Looking at the mean over all users (counting each user's results with equal weight), 1.43 keywords were used per search with a query depth of 1.07. While comparisons with the desktop model are difficult[4], we believe that in the average case it requires more than 1.07 "operations" to locate a piece of information within a desktop file and folder system.

6.7.3 Substream Accuracy

One of the more crucial variables is the success rate of the substream operations. Rates varied from 78.2% to 94.7% over the users (Table 6.6), and the advanced users had higher success rates than the clerical workers. One possible explanation is that the advanced users had used Lifestreams for a longer period of time before the testing period (mean time of 12 versus 2 months).

Given that our measure may include error (as we pointed out in our heuristics for determining success rates), we asked the users in the survey

```
Were you usually able to find what you were looking for?
If not, was the system frustrating in this respect?
```

Representative responses include:

> *Pretty much always able to find what I was looking for — and quickly.*
>
> *(I) was able to find what I wanted.*
>
> *Yes. (I) hardly ever (had trouble).*
>
> *...substreams did not always contain exactly all the relevant information I wanted. It was easy enough to find a simple document, but not a complete collection.*

[4] We've found no empirical data on "navigation" depth in desktops.

Usually — concept and category-based searching would help though.

The responses indicate that while users could locate what they wanted to, nevertheless the prototype could be improved with more powerful search capabilities and with improved ability to create substreams by instance.

6.7.4 Substream Size

We also logged the size of the resulting substreams. Of the 207 substreams created, the minimum/median/mean/maximum substream sizes were 1/15/89/2824 (σ = 3.8). 59% of the substreams were smaller than 25 documents, 71% were smaller than 50 documents, 86% were smaller than 100 documents and 92% were smaller than 200 documents. Only 11 (5%) exceeded 500 documents. Another way to view this data is in the form of a bar chart (Figure 6.1). Here each bar represents the number of substreams (from the 207 find operations) that have a size that falls into the corresponding interval. Note that (with the exception of the first two bars) each successive bar has an interval that is twice that of the previous bar.

One might argue that substream sizes will uniformly grow along with the stream size. If so, the average stream size for these queries is roughly 7000 documents, which means that 59% of the substreams contained a mean of roughly .015% of the documents in the main stream. If we extrapolate to a stream size of 100,000 document then the mean substream would be 1500 documents in size. However if we examine the median and mean sizes of the substreams across all users[5] (Table 6.7) compared to their lifestream's size we find that as we step up in stream size from 1400→3200→6900→16700 the mean substream size (as a percent of the main stream size) goes from 2%→1.2%→.8%→.6% and the median size from .4%→.2%→.2%→.1%. While this is encouraging, the falloff in growth rate of the substreams is still too slow to prevent a slow rise in the size of substreams.

There is, however, a compelling reason to think this will not be a problem in the general case because we have chronology to rely on; that is, since the value of information is likely to decrease over time the user is more likely to find the information near the head of the substream or to scan back to the appropriate time in the past (and we have seen from Lansdale's work that users are good at remembering when information was created). Queries in future Lifestreams systems may also allow users to constrain the queries to the last few years or months and effectively cut the stream down to a reasonable number of documents.

6.7.5 Substream Response Time

In Chapter 4 we discussed Shneiderman's concept of direct manipulation and the importance of allowing the user to carry out tasks quickly and observe results immediately.

[5] We are making the assumption that the average substream size (relative to stream size) is somewhat uniform across users. Obviously, this may or may not be the case and requires further study.

	G_2	G_1	C_1	C_2
Stream Size	16700	16700	3200	1400
Minimum	1	1	1	1
Median	21	12	9	6
Mean	106	53	41	40
Max	2824	1550	529	138
Standard Deviation	7.0	5.9	8.1	15.2

Table 6.7: Size of Substreams.

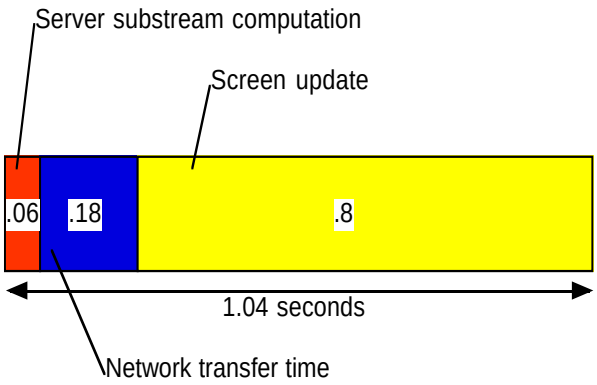


Figure 6.2: Average time to compute and display a substream.

Using the timing results from all 207 find operations we found that it takes an average of 1.04 seconds for Lifestreams to compute a substream and display it on the screen. Of the 1.04 seconds (see Figure 6.2) it requires (on average) .06 seconds for the server to compute the substream, .18 seconds to transfer data describing the substream to and from the server, and .8 seconds to display the substream on the screen. One second is twice as long as Shneiderman's "optimal" update time of 500 ms but given that 77% of the substream time is being consumed by screen update there is much room for improvement (recall that our client is written in Tcl/Tk, a scripting language, not a compiled language). In any case, the users were content with the response of the system as indicated in the QUIS questionnaire and the survey.

6.7.6 Substream Management Styles

We have observed a variety of "styles" in which substreams are managed by users. Some users maintain a small number (typically less than a dozen) of substreams which they revisit from time to time.

	G_1	G_2	C_1	C_2
Substream accesses	78	21	4	6
Hits	66	19	3	4
Success Rate	85%	90%	75%	66%

Table 6.8: Locating information through persistent substreams.

> *Mostly I remove (substreams) except for the 2-4 I always keep around.*

Users C_2 and G_1 fell into this category, both creating a handful of substreams that they maintained. These subjects still used the find operation to locate "short-term" information, but they quickly removed those substreams.

Other users tend to always use the find operation to locate information, even if they already have an existing substream which was created with the same search query. These users may allow a large number of substreams to build-up before removing the whole lot. Before the logging period began user C_1 was found to have over 100 substreams that he never reused. When asked in the survey if he kept substreams around for future use he responded:

> *No I don't. I remove them when a lot have accumulated ... [the] user should be able to set a default where substreams automatically "self erase" after a day or a user specified period of time.*

These styles can be seen in Table 6.8 where we list the substream operations each user performed. User G_1 claimed to reuse substreams while user G_2 claimed to almost never reuse them; consistent with this user G_1 revisited substreams nearly four times the number of times G_2 did and (referring back to Table 6.5) G_2 used find 40% more times than G_1. Although there is sparse data from the clerical workers the same trend seems to occur with C_1 using existing substreams at a higher rate (comparing finds) than C_2.

6.8 Summary

With two exceptions (with respect to the reliability of the prototype), our initial user pool responded in a highly positive manner across all QUIS questions (mean over all questions = 8.0/9). Learning had the highest subjective satisfaction score of all QUIS categories indicating that Lifestreams has a short learning curve. Users also indicated that they found the Lifestreams metaphor and interface an easy and intuitive way to

manage online information. Data from instrumentation suggests that substreams were a effective and efficient mechanism for locating information.

We also learned that users felt they would benefit from improved and more flexible searching capabilities and the ability to supply time-intervals for substreams. Advanced users also wanted shortcuts to minimize the number of steps required for an operation. With respect to the interface, users felt the display could have been more information rich and, from the response of novice users, we learned that a minimal interface is not always better — often users want several repetitive commands wrapped up into one button or function. By packaging operations (such as Clone-Edit-Xfer into a Reply button) we reduce the number of steps required to perform a common operation.

Finally, in the survey we asked the subjects whether or not they would continue to use Lifestreams if it were a robust and supported system. All users responded in a positive manner. In fact five of the six subjects responded in a very positive manner[6]:

Yes!

Absolutely!

Yes!

Absolutely

Definitely

[6]The sixth subject, responded *"Yes, assuming it was faster and a little more refined."* This user worked with the system before the summer of 1996 (before some performance problems were fixed and before interface improvements were made).

Chapter 7

Information Management Revisited

We now return to the user studies of the information management community that we introduced in Chapter 1 and examine them in more detail. In the process we will explore the Lifestreams model in the context of these studies. First we examine classifications of user information types developed by Cole and Barreau & Nardi. Next we revisit the work of Malone and examine his specific recommendations for electronic systems. We then examine user studies that resulted in specific task classifications. Finally, we revisit Lansdale's work.

7.1 Information "Types"

The user studies of Cole [Col82] and Barreau & Nardi [BN95] examined the ways users of desktop systems use, process and manage electronic information. Both studies resulted in valuable classification schemes for electronic information. Barreau and Nardi found that users commonly deal with three types of information:

- *Working information* is usually relevant to long or medium term tasks that the user is working on actively. Typically it has a "shelf life" (period of use) of days, weeks or months and is important enough to be stored in its own folder or location. Users worked with this information frequently enough to know exactly where it was stored.

- *Archived information* has a shelf life of months or years, but is infrequently relevant to the user's current working tasks.

- *Ephemeral information* has a "short shelf life" and is only used for a short amount of time. On desktop systems users tend to place these items on the desktop and later discard them. Ephemeral information may include todo list items,

reminders, or electronic mail. Note that this includes "reminding" types of information.

Cole found practically the same categories:

- *Personal work files* refers to information that is categorized according to some user-defined strategy and is relevant to the user's ongoing work activities.
- *Archived storage* refers to structured storage of no direct relevance to a user's predicted work schedule.
- *Action items* consists of information that is immediately important or needs to be dealt with in the near *future*.

We have already discussed in Chapter 1 the way in which current desktop systems fail to support these types gracefully, but these categories make it easier to see why: "working information," for example, requires that the user maintain disciplined filing practices, something which, according to previous work, users find difficult, especially over time. Current systems also force the job of maintaining organization on the user rather than relying on the inherent content of the information. With respect to "archived information" desktop systems provide little support for the archiving or retrieval of older information. The burden of both are forced on the user, who is expected to develop complex archiving schemes (here we mean archiving in the conceptual sense — users don't have to worry about old information cluttering their desktops or getting in the way) and remember them (for retrieval) over time. We have seen the results: users often throw out information rather than be burdened with its upkeep. "Ephemeral information" or "action items" quickly build up into piles of information that have to be searched with brute force methods [Coo95]. This type of information is often the source of "messy electronic desktops." A major portion of ephemeral information includes items of a reminding nature, yet as we have argued electronic systems provide no integrated support for convenient or effective reminding.

Let us now turn to Lifestreams and study the ways in which it accommodates these classifications. A lifestream can be divided into three portions: *past*, *present*, and *future*. The present portion normally occupies the head of a stream or substream. The future portion is distinct and usually hidden from view (i.e., in our prototype), and the past portion recedes into the distance as documents are added to the stream. These stream segments closely mirror the information categories. The "present" portion of the stream holds "working documents;" this is also typically where new documents are created and where incoming documents are placed. Substreams are used as working areas in a manner that is similar to folders. A substream can hold the documents relevant to a particular working task, allowing the user to locate and filter out extraneous information quickly. As documents age and newer documents are added, older documents recede from the user's view and are "archived" in the process; if at some future

point they need the archived information, it can be located with `find`. Ephemeral information passes through the present and into the past, receding out of the users view in the process—as in a quick response mail message (e.g., "lunch?") — yet it remains available for for retrieval if needed.

"Future creation" provides a natural method of posting reminders and calendar information by allowing the user to dial to the future and deposit a document there—say, a meeting reminder. This facilitates reminding in two ways: the user can use the stream as a todo list by dialing to the future at any time to see the reminding items on his stream or he can use future documents as truly effective reminders—when the creation date of a future document arrives, the reminder automatically appears in the present and alerts the user (through the same mechanism that alerts of the user of a new document on the stream, such an incoming email message).

7.2 Malone Revisited

While Malone [Mal83] did not develop task classifications in his work he did use the results of his study to suggest avenues for the automation and simplification of electronic systems. Malone chose two specific areas for these recommendations: finding and reminding.

7.2.1 Finding

Malone suggests that electronic systems can help with "finding" in three areas: creating classifications, classifying information and retrieving information. Creating classifications, Malone comments, is nontrivial and a "major barrier to keeping file systems current." To improve information classification Malone recommends incorporating three capabilities into computer systems:

- *Multiple classification* allows a document to be easily put into several categories.

  A Lifestreams document can be located in as many substreams as necessary, or to none.

- *Deferred classification* allows users to store information in the same physical location without having to "title" the information explicitly.

  Lifestreams uses chronology as a basic storage model, removing the need for explicit storage or naming.

- *Automatic classification* allows the computer to do as much classification as possible based on fields or attributes of the document. Malone notes that chronology is one useful method of automatic classification and suggests that users may want to "rewind or fast-forward the (history of) their desktop to locate the last time the desired document was on their desk."

> Persistent substreams provide automatic classification as new documents are added to a lifestream. Lifestreams, of course, makes use of chronology as a basic storage model.

Last, Malone notes that in retrieval, current systems only allow users to specify one retrieval key (e.g., the name of a file) at a time. He suggests that systems should allow users to use more than one retrieval key at at time, such as the document's type, time of creation and information about its content. While the situation has changed somewhat since Malone's time, nevertheless desktop systems can still be improved. Lifestreams allows retrieval on all these cues. Users can specify file types, file content and browse through information based on chronology. With respect to chronology Malone suggests that chronology actually be used as a search method (e.g., find a document created last week). In Lifestreams, this aspect is used for browsing rather than as an explicit search term, although future Lifestreams systems can easily and naturally accommodate time-based values in searches.

7.2.2 Reminding

In the area of reminding Malone suggests that systems should make it easy for users to "store certain information so that it will automatically appear, without being requested." Lifestreams provides this in a natural way by allowing users to create documents in the future. In addition, our research prototype provides an effective means of allowing the reminders to "automatically appear" at the time the user needs to be reminded.

7.3 Task Analysis

We now look at work where users were observed and the tasks they typically perform were categorized.

7.3.1 Whittaker and Sidner

We first examine a large-scale user study conducted by Whittaker and Sidner [WS96]; in this study they examined how users make use of electronic mail to perform a variety of activities beyond its initial role as a means of asynchronous communication. This study is important not only because users spend a significant portion of time using electronic mail [Gat96, WS96] but because Whittaker and Sidner specifically studied the ways in which users managed their electronic information with the goal of discovering how email might fully support *all* the electronic tasks a user performs. They discovered three main functions that users perform: *task management*, *personal archiving*, and *asynchronous communication*. Whittaker and Sidner describe task management as follows:

> "*Task management* requires users to ensure that information relating to current tasks is readily *available*. This both preserves *task content* and allows users to determine the progress of ongoing tasks. Task management also involves *reminding* oneself about when particular tasks or actions have to be executed."

Personal archiving:

> "*Personal archiving* or *filing* addresses how people organize and categorize longer term information, so that it can later be retrieved. *Archives* are not of immediate relevance to current tasks, but are constructed for reference or anticipated future use. Research shows that users experience major problems in generating appropriate folder labels when filing longer term information for later retrieval, and in reconstructing these labels when they engage in later retrieval."

Asynchronous communication:

> "*Asynchronous communication* is concerned with the interaction in a permanent medium across space and time... Such interactions are seldom one-shot, and workers often engage in multiple intermittent interactions in order to complete a task. Workers are also usually engaged in several independent, but concurrent ongoing conversations, with the requirements of tracking separate conversational threads and switching contexts between conversations."

Their primary finding was that email was being used for purposes beyond asynchronous communication (its original, intended purpose) because of a lack of support for task management and personal archiving in their desktop systems.

Let us now consider these important tasks in the context of Lifestreams. Lifestreams includes asynchronous communication as a basic functionality of its model (through the `transfer` operation). We discuss this aspect of Lifestreams in detail in our focus on email later in this chapter.

Whittaker and Sidner's "task management" requires that information be "readily available." Lifestreams provides this by allowing the documents in a user's lifestream to always remain available for search (whether or not it has already been organized in other substreams). In this way substreaming provides a more flexible means of task management than rigid directories. This is best demonstrated through examples like Lansdale's multiple projects scenario presented in Chapter 1; while desktop systems lock users into file organizations, Lifestreams allows a fluid way of organizing information that can change along with users' needs. Whittaker and Sidner's "task context" is provided through persistent substreams, analogous to a directory that can maintain a collection of documents over time. Because substreams actively capture new content,

the user can more readily "determine the progress of ongoing tasks." Moreover, substreams can contain all the information related to ongoing tasks (reports, email, etc.); we will return to this point in the next section. Whittaker and Sidner also explicitly point out the importance of reminding to task management — we have already described how Lifestreams supplies reminding in a natural manner (as we saw from user comments in Chapter 6).

On the topic of personal archiving, Whittaker and Sidner point out that "users experience major problems in generating appropriate folder labels when filing longer term information for later retrieval" and in "reconstructing these labels when they engage in later retrieval." Lifestreams provides a solution to these problems in two ways: (1) through the time-ordered stream model documents gradually recede from the users "view" and are conceptually archived in the process (2) all documents in Lifestreams remain available for retrieval via content. In this way users never have to invent "folder labels" or remember them; they only have to remember the content of the information or when it was created or some combination of the two (something users are good at [Lan88b]).

7.3.2 Erickson

We now examine a second long-term study at Apple Computer by Thomas Erickson. Erickson developed and analyzed his own use of a personal electronic notebook over a three year period [Eri96]. The notebook, implemented in Hypercard on an Apple Powerbook, was an attempt to develop a system that mirrored the everyday tasks of Erickson (a casual user). During this work Erickson aimed to understand the features and the modes of use that made his notebook useful. This work is particularly germane because it studies a user in the context of a personal information system.

Erickson primarily used his notebook in a work setting to manage tasks, record notes, compose and send electronic mail and as an archive for his information. He used these various functions in an iterative manner. Usually at the beginning of the day he checks and updates his ToDo list (task management) and then checks email and his calendar (which may result in further alterations to his ToDo list). Throughout the day he iterates through making notes, checking his ToDo list and using electronic mail.

Less often Erickson performs the following tasks: searching through his information for a particular item, browsing the last month of activities and incorporating it into a summary, or transcribing something written into his Powerbook. Erickson had originally envisioned that he would link and develop cross references between information in his collection of documents, but he quickly abandoned this idea because it caused disruptive context shifts. In a comparison with Erickson's old paper-based method of organizing information, he found that his electronic notebook had the following effect: he took more notes because, through the searching and browsing capability, he was able to find old information more readily (and it was more legible than in handwritten form).

In summary, Erickson uses his notebook primarily for managing his tasks and schedule (ToDos and calendar), creating and managing notes, performing messaging through electronic mail, and archiving and later searching for information.

How do these tasks of a typical user of a personal information management system fit to Lifestreams? As we've seen Lifestreams accommodates task management by allowing the user to create task contexts and also through managing reminders. In previous chapter we presented several methods used in Lifestreams to manage Todo lists. Lifestreams provides a natural "paper-like" system for creating notes where neither names nor storage folders are needed, and, as in Erickson's system, where notes can be searched for[1].

Erickson makes an interesting point about the combination of messaging and note making in his system:

> "(My system) is useful because of the synergy between note making and messaging. As noted, the messaging built into (my system) increases the potential utility (and quality) of note making, and the note making in turn provide grist for messages and other re-uses of content."

This is interesting because Erickson's system did not have a fully integrated communications system. He moved text back and forth between HyperCard and mailer. Lifestreams should make the synergy stronger with its built-in transfer operation that allows any document to become a mail message.

Erickson's last task, archiving and retrieval, has been discussed at length with respect to Lifestreams. Erickson does comment on some interesting aspects of searching when the underlying system stores a user's entire information collection (that is, all notes, ToDo lists, email, etc.):

> "Browsing itself is made easier because the [information that is] normally ephemeral, ... is captured – ToDo lists, email messages, notes – [and] provides more cues about what is being searched for."[2]

Two factors in the Lifestreams system help to preserve these cues: (1) all information in your electronic life is stored in your lifestream (2) chronological storage groups these items together in the manner they were created. This supports our argument in Chapter 2 for chronological storage.

Erickson also points out that search is useful when "trying to put together a summary of what has happened on a project." Here is further evidence that users are in need of a summarize capability (as we have in Lifestreams).

---

[1] Lifestreams improves on the technology of Erickson's system because we use an indexing system (he relied on the internal, non-index search capability of HyperCard). In addition Erickson couldn't use substreams to organize information.

[2] Although not chronologically-based, Erickson's system groups daily items within one page and provides a table of contents.

Last, Erickson states that his system is "good at dealing with text, and the many ways in which I reflect, communicate and act are entwined in textual representation." We believe the same is true of most users and we have shown how Lifestreams also reflects this style of work. Erickson concludes that "probably the most significant impact of (his system) is a sort of synergy among the activities of note making, reference, and message." The parallel with Lifestreams should be clear as it is, in its core, a note making, reference, message and reminding system.

7.3.3 Tasks from the Knowledge Navigator

Finally we take a look at tasks not from a user study, but from a prototype system. In 1987 Apple Computer created the video "Knowledge Navigator," which presents a futuristic prototype of a "conversational computer" [Com87]. While the video has drawn much interest because of the conversational aspect of the system, the information tasks the user is fluently carrying out are more enlightening. We present the dialog from the first half of the video in Figure 7.1, where we join a professor as he "checks in" with his computer.

Setting AI issues aside, we examine this interaction in the context of information management. In `1` the computer begins by summarizing messages that have arrived since the user last spoke with the computer — an operation that could (in theory) be accomplished by a Lifetreams' summary of new messages. Then, in `2`, the professor immediately creates a reminder. Next, in `3`, the computer summarizes the reminders on the future of the professor's "stream." In `5`, the professor asks the computer to find information based on content and chronologically. The computer presents the results and the professor, not happy, asks the computer to find all articles he has not yet read — an attribute based search. The computer then asks for clarification and summarizes the search results in `9`. In `10-11` the professor performs messaging with the computer's help. In `12` the professor performs another chronology and content-based find and the computer responds with a summary.

The Knowledge Navigator provides an excellent example of a fluid software environment that mirrors the user's work habits. We note that the primary operations performed in the video (summarize - remind - find - find - summarize - messaging - find - summarize) directly correspond to the primitive operations of Lifestreams (leaving aside new and clone).

7.4 Lansdale Revisited

Finally we revisit the examples of Lansdale in the context of Lifestreams. Our first example illustrates a user who needs to retrieve information in a manner that is different than the way it was organized within a hierarchy.

Professor enters room

1. **Computer:** "You have three messages: your graduate research team in Guatemala just checking in; Robert Jordan, a second semester junior, requesting a second extension on his term paper; and your mother reminding you about your father..."

Professor interrupts

2. **Professor:** "surprise birthday party next sunday"
3. **Computer:** "Today you have a faculty lunch at 12:00, you need to take Kathy to the airport by 2:00, you have a lecture at 4:15 on deforestation in the Amazon Rain Forest."
4. **Professor:** "Right."
5. **Professor:** "Let me see the lecture notes from last semester."

Computer displays notes.

6. **Professor:** " No that's not enough, I need to review the more recent literature. Pull up all the articles I haven't read yet."
7. **Computer:** "Journal articles only?"
8. **Professor:** "Fine."
9. **Computer:** "Your friend, Jill Gilbert, has published an article about deforestation in the Amazon and it's effects on rainfall in the sub-Sahara."

Computer continues with summary of Jill's work.

10. **Professor:** "Contact Jill."
11. **Computer:** "I'm sorry she's not available right now. I left a message that you had called."
12. **Professor:** " Ok, let's see. There's an article about 5 years ago, Dr. Flempson or something. He really disagreed with the direction of Jill's research."
13. **Computer**: "John Fleming, of UpSilla University."

Computer continues with summary of his work.

14. **Professor:** "Yes, that is it."

...

Figure 7.1: Dialog from Apple's "Knowledge Navigator".

> "My boss wants to see all the project reviews I have carried out over the last six months. The trouble is, they are filed under each of the individual projects. It will take me ages to work through and dig them all out."

In Lifestreams the user can perform a find operation to retrieve all "project reviews." The result of the find is a substream that acts as a temporary context for his task. With the assumption that his Lifestreams system is equipped with a method of summarizing text files[3], he could use the summary operation to condense the project reviews into one document, from which he could prepare the report for his boss. Taking this one step further, he could then transfer the report to his boss' stream in a third operation.

Users often remember chronological information about documents (discussed in Chapter 1) as seen in this example:

> "Yes I remember that paper. It came at the same time as the product audit. I can't remember what happened to it, though."

In Lifestreams the user first substreams to locate the product audit and notes the time of its creation by looking at the glance view of the document. He then returns to his mail stream, zooms back to that time in the past (in our interface via the time-enhanced scroll bar) and browses for the document. If he happens to remember something about the paper (it mentioned "Schwartz"), then he can first create a substream based on that information and then zoom to the past. This reduces the amount of information the user needs to peruse.

Often a user's memory of where things are filed breaks down. In this case users rely on the mnemonics of file names and directory names to locate information. Unfortunately, as Dumais [DL83] pointed out, over time these categories overlap and become ambiguous.

> "The document I want is the French Finance Committee's minutes, but I've tried looking under 'French', and 'Committees' and it's not there. Perhaps it's under 'France'. I don't know I may as well search through the whole lot."

Searching through the whole lot is obviously a pain in most desktop systems. Even with a search over file names it is not clear the user will find the information. However with Lifetreams' content-based search the user can often locate the information in one step (e.g., with a search for `"(French or France) committee minutes"`). Here the burden of finding the information is placed on the computer, which maintains in data-structures the content information needed for retrieval. In desktop systems the organization structure is maintained in the users head.

[3]Summary engines such as Apple's Vespa Linear lead us to believe this is not a big assumption.

7.5 Summary

The work of Malone, Lansdale and others suggest avenues for the simplification and improvement of electronic systems. Other work, such as Whittaker and Erickson's, produced valuable classifications schemes for information types and tasks that can be used to examine the capabilities of current desk systems and suggest how future systems might better serve us.

In this chapter we have examined these findings and explored how Lifestreams as a model incorporates many of these suggested avenues and handles user information types and tasks in a more elegant way than current desktop systems. We now move on to examine Lifestreams in the context of related software systems.

Chapter 8

Related Work

Lifestreams builds upon previous work in a number of diverse areas: personal computer indexing systems, information retrieval and filtering systems, corporate document and archiving systems, personal information handlers, time and contact managers, workflow systems, financial managers, and scheduling systems. In this section we survey the landscape, comparing and contrasting representative works from each area with Lifestreams. We conclude this section with a comparison of features from related systems and Lifestreams.

While we do not claim the systems we visit in this chapter are all inclusive, we do believe they are representative of the landscape of existing systems. For instance, there are several "disk indexing" applications on the market and available as shareware, however, we primarily cover On Technologies' product, as it has the same functionality as other applications on the market.

8.1 Information Retrieval Systems

Historically, work in information retrieval (IR) has included both information retrieval and information filtering. Information retrieval concerns itself with the efficient storage and retrieval of documents, in most cases text-based documents. Filtering is a variant of information retrieval that typically targets real-time streams. IR techniques have traditionally been used on large centralized collections of documents, such as the Nexis/Lexis database. More recent work addresses how such techniques might benefit everyday computer users with their own document collections. We present examples of both information retrieval and filtering here.

8.1.1 WAIS

The Wide Area Information Service (WAIS) is a commercial-grade information retrieval system [Kah91] that provides access to an arbitrary number of network document col-

lections. WAIS users formulate search queries over any number of document collections and can refine their searches using relevance feedback. Queries can be saved, and the retrieval process automated (e.g., retrieval can be scheduled to occur at periodic intervals), allowing WAIS to function as a rudimentary "clipping server." WAIS provides a client/server architecture for accessing multiple document collections as well as the use of relevance feedback to guide user searches.

In comparison to Lifestreams, WAIS is used to search centralized document collections rather than personal information. The ability to save and reschedule searches has a tenuous relation to substreams but WAIS does not provide builtin communication or summarizing capabilities.

8.1.2 Tapestry

Tapestry, a research system developed at Xerox PARC [GNOT92], was constructed to manage a large number of incoming articles, such as netnews and electronic mailing lists, for a workgroup. As new articles arrive in Tapestry they are collected and archived within a global database—like Lifestreams, they are meant to be archived indefinitely. Rather then perusing the collection of documents directly, users supply content-based filters that are installed in the system and then iteratively applied to the database. Successfully filtered documents are then forwarded to each user's mailbox where an "appraiser" may do additional filtering (such as sorting the mail into folders) before the user sees the list of new documents. Like Lifestreams, Tapestry also infers a number of mail-related attributes for each document which can be used in filtering. One especially novel aspect of the system is its support for "collaborative filtering" whereby group members can recommend an article as worthy (or not) of being read by others.

Like Lifestreams, Tapestry allows the user to construct filters that continually filter incoming documents; this combined with an appraiser that categorizes documents into folders gives us something that looks very much like a substream. It's not a substream though, because a substream creates a virtual collection of documents based on a specific filter, whereas Tapestry folders are the result of many filtering operations merged together. The main difference between Tapestry and Lifestreams is one of philosophy; Tapestry is not a system for managing personal info chunks, it's more of a global filtering engine for all the information coming into a workgroup. While Tapestry provides some Lifestreams type mechanisms, it directs them at objects incoming from the outside world and not at the user's filespace.

8.1.3 MIT Semantic File System

The MIT Semantic File System [GJSO91] provides associative access to a file system via *virtual directories*. Using native directory commands (such as `ls` and `cd`), virtual directory names are interpreted as associative queries. The results of a query are

computed via an automatically indexed set of attributes (field/value pairs). This index is generated by a number of *transducers* that map files of specific types (e.g., C files, TEXfiles, etc.) to a set of attributes.

The Semantic File System is novel in its ability to describe a desired view of the file system's contents. This description maps to no actual folder or directory of information but to a virtual one computed on demand. Indexing is important because it guarantees acceptable response time on queries (we established this in Chapter 3). Indexing also enables searches on a file's entire content.

The Semantic File System shows reasonable performance on a realistically sized file system with queries answered in the one to two second range. The results of the work "suggest that semantic file systems can be used to find information more quickly than is possible using ordinary file systems."

The connection with Lifestreams is clear: both provide virtual organization documents through search. Unlike Lifestreams, the Semantic File System does not provide an integrated information management environment, just a search tool. However, the semantic file system suggests improvements that can be made to the Lifestreams system via transducers. Today, Lifestreams only indexes text documents. With transducers, other document types can be indexed and the overall method of indexing improved.

8.1.4 Glimpse

Glimpse is an indexing tool for Unix files systems developed by Udi Manber and Sun Wu at the University of Arizona that manages personal information and uses an indexing approach that fits in-between inverse indices and signature files [MW93]. This approach produces index files of only 2% to 4% of the original text size (versus 50% to 300% for traditional techniques) while providing average search times on the order of a few seconds; Manber and Wu argue that this is sufficient for personal information. Glimpse achieves small indices by breaking text files into blocks and adding one record to the index per occurrence of each word in the block (as opposed to storing every occurrence). Searching is a two phase process that uses *agrep* [WM94]. *Agrep* is a generalized version of *grep* that allows the user to specify a number of errors which can be insertions, deletions, or substitutions. This allows Glimpse to handle misspellings and other common search mistakes. In Glimpse, *agrep* is first used to search the index itself, and then *agrep* uses the output records to perform a search on the actual files.

Glimpse is similar to Lifestreams in that it allows quick access to a users information via a search mechanism. Using Glimpse however is a "one shot" process that does not allow the creation of virtual directories. In short, Glimpse is primarily an extension of search tools such as *grep* (albeit an important extension allowing fast access to a user's total information store) and not an integrated work environment. Work on Glimpse is encouraging however, and suggests an intriguing method of cataloging and search large personal data collections.

8.1.5 Apple Find

The Apple "Find" application is often thought of as a last resort in locating files on the Macintosh. For many, who are not as organized as they would like to be, Find becomes a way of life. Find presents a simple interface to the user that allows the entry of keywords that are matched to filenames on the user's hard drive. In the newest version of the Macintosh operation system, System 7.5, Find has been extended to allow for custom searches to be done on file type and dates of creation, and so on.

The similarity with Lifestreams is in the mode of operations: using search rather than using locale to find dated information. Find differs in many ways however, Find searches all information at run time, it doesn't index disk information incrementally. It also doesn't create virtual directories based on the search criteria (although the system 7.5 presents the information as if it is a virtual directory).

8.2 Database Management Systems

Lifestreams shares several ideas with the database management systems (DBMS). Lifestreams consist of a number of documents that can be described in a record-like manner. Substreams are related to "views" in relational databases [Dat86]. Future documents are related to "triggers" (in Lifestreams, the trigger occurs when the creation date of the document slides into the past). There are also connections between Lifestreams and temporal databases [Sno90], temporal logic [All83], and sequence database systems [SLR84] where time and/or logical sequences play a crucial role in the system.

Lifestreams differs from previous work in the DBMS communities in several important ways. Lifestreams is first and foremost a system for personal information management, while DBM systems are used for centralized data collections. Lifestreams also operates on a flexible data model that includes many types of media and is based on flexible indexing and retrieval techniques. Lifestreams' search is provided by a simple system of keywords and boolean operations that allows everyday users to create queries. Database systems rely on query languages such as SQL, which are not accessible to the typical computer user. Last, Lifestreams provides a novel user interface to a database system, which has been called for by the leading researchers of the DB community [SSU95].

8.3 Personal Information Managers

There are a number of personal information management (PIM) packages and platforms on the market. Like Lifestreams these packages are suited towards managing personal information such as contacts, ongoing tasks and schedules. In this section we first cover the most common types of software packages, and then the Newton platform which subsumes the functionality of most PIM devices. Last we cover an interesting

shareware application with a philosophy similar to that of Lifestreams.

8.3.1 ToDo List Managers

ToDo List managers, such as LandWare's ToDo List [War95], all provide the same basic functionality, that of managing a simple list of personal tasks. ToDo List allows the user to create a number of time-stamped documents (of type text or voice memo) and displays them according to their date and an optional priority (priority items are underlined). Additional information can be displayed along with each item, such as the number of days the task is overdue. A click box is provided for the user to indicate the task is finished. ToDo List also allows the user to search for text within the ToDo list text documents.

As we have already mentioned in Chapter 5, the todo list capabilities are a simple "add on" to the Lifestreams system, as the stream provides a natural data structure for todo list applications. The todo items are also made available along with the rest of the user's documents. Moreover, the rest of Lifestreams' capabilities are not a simple "add on" to todo list managers.

8.3.2 Contact Managers and Time Trackers

In their simplest form contact managers store and provide access to names, addresses, phone numbers, and other information about a number of (typically business) contacts. Taken to their logical conclusion, contact managers retrieve phone numbers, dial phones, log calls, automate the sending of electronic mail, print envelopes and automate letter sending. Many contact managers also include "time tracking," the ability to track "billable hours" by having the computer log and charge standard hourly rates for the duration of some task. Reports and bills can usually be generated from these logs.

As with the todo list, we mentioned in Chapter 5 that these capabilities are simple "add ons" to the Lifestreams system.

8.3.3 The Newton

The Newton represents an "architectural" approach to implementing a personal information manager as it includes both a novel hardware and software design. Since the Newton includes many of the PIM features we have already discussed, we will present only its novel features here.

We have introduced the the Newton in Chapter 4. Here we will touch on the more interesting aspects of its software architecture. The most interesting aspect of the Newton is its storage model. A Newton can have any number of *stores*; stores usually represent physical storage such as RAM, a memory card, or a hard drive. Applications access stores by creating, reading, and writing *soups* [Com93]. Soups are somewhat analogous to files; they are persistent storage structures that can be accessed by any

process. Soups are the primary organization scheme for the Newton, which contains no hierarchical storage model. Users typically search for information rather than statically organize it.

Each soup contains entries that are made up of typically homogeneous data structures, with zero or more indexed fields. For each soup, applications can define a search interface that allows the Newton's personal assistant (a software functionality available to the user) to search for arbitrary information. For example, if the user tells the personal assistant to find everything it knows about "Bob Schwartz" then an arbitrary application can have its data included in the search.

Lifestreams and the Newton are similar in that they use a non-hierarchical storage model (soups versus the time-ordered stream) and prefer searching to static organization. They also both include task automation (Lifestreams uses personal agents, and Newton uses its personal assistant). They differ in that the Newton doesn't allow organization schemes, like substreams, to be created, nor does it provide a default organization scheme for all information. We have suggested in [Fre95] how the Newton might be altered and benefit from a Lifestreams like model.

8.3.4 Guy Friday

Guy Friday is a shareware application developed by former Yale undergraduate Matthew Klein on the Macintosh platform. Guy Friday is an attempt to develop a personal information manager for unorganized people. Guy Friday was designed to work like scraps of paper; the user simply jots down notes, phone numbers, etc. on electronic notes (called "nuggets"). Notes are displayed in an arbitrary and overlapping manner on the user's screen. To recall a note, the user enters keywords into a dialog box and the relevant notes are displayed on the screen.

Guy Friday includes two other features, phone dialing and reminders. Phone dialing works by searching the top-most (selected) note for a phone number and dialing it via the computer. Reminders are added via a dialog box that ask for a text message, a date and time, and whether the reminder should occur once, daily, weekly, monthly or yearly. When the reminder expires, a dialog message is displayed with the text of the reminder.

Guy Friday and Lifestreams both include a simply way of creating and filtering information as well as phone dialing and reminder functions. They differ in that Guy Friday contains no organizing framework other than quick searches. They also differ in that reminders in Guy Friday are a separate aspect of the system, discrete from the notes. That is, a nugget can't be a reminder. Reminders can be created and listed, but do not inhabit the same space as notes and can't be searched for.

8.4 Schedulers and Meeting Makers

There are a number of scheduling applications on the market (such as On Technologies' Meeting Maker XP [Mee94]) that provide the same basic functionality: electronically scheduling a meeting of n people. The meeting maker scenario is usually: (1) someone proposes a meeting to the meeting maker along with a list of required participants (2) the meeting maker (usually via integrated email) asks each participant to accept or reject the proposed meeting, and (3) the meeting is then either scheduled or canceled (or perhaps an alternative suggested and a second round begun). The sophistication of these systems varies but almost all attempt to at least examine existing schedules to avoid scheduling conflicts. Some systems include complex rule-based logic for proposing meeting times/places and many systems provide electronic reminders that can be sent out before a scheduled meeting.

The scheduler or "meeting-maker" is perhaps the canonical example of an important functionality (typically implemented in a standalone application) that can be incorporated into Lifestreams. Lifestreams' agents along with its time-based stream and integrated email provide a natural setting for implementing schedulers.

8.5 Corporate Document Systems

Corporate document systems fall into the categories of document archive systems, "groupware" systems, and workflow technologies (the automation of conventional "paper-trials" with electronic systems). Our previous discussion of information retrieval covers the topic of document archive systems, although there are some corporate retrieval systems that include workflow and other enhancements. In this section we first cover the predominant groupware platform on the market, Lotus Notes, and then discuss workflow in general terms.

8.5.1 Document Systems: Lotus Notes

Lotus Notes^{TM} is perhaps the most successful "workgroup" communications product on the market. Notes acts both as a company document archive/database and also facilitates "workgroup" communications among employees. There are several analogies between Lifestreams and Lotus Notes: Notes maintains document collections in databases and allows custom viewing and filtering of a particular database via a "view" (a view is a specification for how to display the documents within a particular database). Notes also incorporates electronic mail as an integral part of the system. On the surface this sounds a lot like Lifestreams, however the domain for which Notes is used is quite different. Why does this matter? Notes is foremost a system for maintaining corporate documents and information—the users don't generally maintain personal documents within Notes. As such, Notes databases are statically configured and, for example,

database filters aren't created by typical employees but rather by the system administrator. In contrast, Lifestreams was created to manage an individual's document collection and allows custom filtering by the user.

Notes provides several methods of creating collaborative structures. The primary method of group collaboration is the "shared database." This amounts to a common area where users can post messages and reply to previous messages. Lifestreams can accommodate the same structure with shared streams (although we haven't explored such use of Lifestreams). Notes also includes a workflow model based on email chains; that is, the originator of a document supplies a chain of people through which the document should pass (possibly getting approval at each point along the way). Lifestreams can be extended to support this workflow model via agents. For instance, the path a document follows can be dynamically supplied by an agent rather than statically supplied by the originator of a message. Configuring Notes for more advanced groupware applications such as scheduling, although not impossible, is a far leap.

8.5.2 Workflow Systems

Because there are a plethora of commercially available workflow systems, with no clear leaders in the market, we discuss workflow systems without reference to a specific product. Several academic systems include workflow capabilities, namely [MGL+87, BR93, Bor93b]. Workflow systems automate information-based tasks within organizations [Bar95]. Workflow falls into several categories: automating administrative tasks (the path of a purchase order through an organization), task monitoring (e.g., how long employee x takes to complete task y), and mediating less structured tasks (draft review by a workgroup).

In Chapter 5, we have suggested how Lifestreams, through its agent support and timed-based storage structure, can be used to support workflow capabilities.

8.6 New Paradigms

A few systems defy categorization (like Lifestreams) and are best labeled new paradigms.

8.6.1 Memoirs

Memoirs is the system that is closest in philosophy to Lifestreams. Developed by Lansdale [Lan88a], Memoirs uses chronology as an underlying storage scheme and search to organize documents. Memoirs uses a "timebase" to display documents in a collection. The timebase displays a sequence of time (the endpoints being determined by the user) along with slots that are highlighted to indicate documents. The user can apply a filter to the timebase, which leaves only relevant slots highlighted, or perform a general search that creates a native Macintosh window (like a folder window) with the relevant documents. Documents are searched via a number of user definable attributes,

such as color (the Macintosh allows the user to label files with color), user-added keywords or assigned icon. Memoirs also includes a separate "diary" which allows the user to move items in and out of the timebase and add reminders.

Lifestreams and Memoirs differ in that Lifestreams fully integrates the ideas of chronology into a single "metaphor," the stream, whereas Memoirs combines a diary with a timebase. Lifestreams also recognizes the essential feature of content-based searching. Without it we are still relying on users to remember categorizations (and create them in the first place). Substreams are incorporated into the Lifestreams model and are not external to the system. Likewise, substreams are "live" and continue to collect documents as they are added to the system. Lifestreams also incorporates communication and the ability to summarize.

8.6.2 Dynamic Queries

Shneiderman's dynamic queries [Shn94] combine direct manipulation and database visualization to allow a user to rapidly filter information through the use of visual components such as sliders and buttons. Manipulation of these components results in the user receiving feedback within 100ms (allowing him to quickly perceive patterns in the data). Visual queries have been applied to a number of domains such as geographic database systems and movie databases. Visual queries have also been implemented in the form of a Unix directory browser [LOS93]. Shneiderman *et al* found that with the browser user queries could be "answered more rapidly because users can filter out irrelevant information and visually scan the remaining information." The location-based alternative (i.e., using the Unix from the command line)[1] "requires more time because users must visually scan a much larger set of information." The browser work is a first step, and as Shneiderman *et al* point out, more work needs to be done integrating visual queries into our day-to-day applications.

Dynamic queries are an intriguing alternative to boolean searches and with improvements in performance could prove an interesting method of searching over Lifestreams documents.

8.6.3 LifeLines

LifeLines [PMR+95] is a general technique for visualizing summaries of personal histories. LifeLines visualizes personal histories by depicting them on a graphical time scale. LifeLines is particularly appropriate for handling biographical data such as medical or legal records. For instance, medical conditions can be represented by timelines and physician consultations can be represented by icons. Plaisant *et al* claim that the natural, time-based ordering "allows comparisons and relationships between the quantities displayed."

[1] Barreau and Nardi describe DOS as a location-based system, we do the same (for comparison) with the UNIX file system.

LifeLines suggests an interesting alternative interface to the Lifestreams system; likewise, LifeLines could use a lifestream as its underlying database.

8.7 Summary — Surveying the Landscape

We now provide a overview of all the systems we've visited in this chapter by comparing them against the essential features we derived in Chapter 1 (transparent storage, dynamic organization, etc.) along with some additional features that are important to the performance and usability of the system; for example, it is not only important that a system allow the user to organize information based on content-based search, but it is also important that the system index the documents in the collection so that search can be accomplished quickly.

Applicability to Personal Use. Is the system suitable for personal storage? Many of the systems we have looked at are intended for group or corporate data, however, as we have seen, these technologies are now being applied to personal information systems.

Searchable Content. Can the data in the system be searched based on its content? As we have seen, systems like the Apple "Find" application allow search, but only by file attributes such as filename, modification dates, and size. In contrast, more flexible systems like the MIT Semantic File System allow searches on document content.

Indexed Content. Given that the system provides searchable content, does the system index the content to improve response time? Previous work has shown that response time is a crucial factor in search-based systems [Shn94]. Indexing content (or similar techniques) is mandatory for search in systems that contain more than one thousand files or document with today's technology (as we discussed in Chapter 3).

Transparent Storage. Does the system support transparent storage? The Newton provides a persistent store and avoids the overhead of naming and explicit storage while the Macintosh uses a conventional file and folder metaphor, which requires naming, the creation of static categories, and explicitly choosing a storage location for every document. So what? Providing transparent storage suggests potential gains for the user, for example, as we discussed in Chapter 1 Malone [Mal83] has shown that the most difficult aspect of managing information is choosing a storage location.

Default Data View. If the system does support transparent storage, does it provide a default way of viewing a set of documents? Our time-ordered stream and Lotus

Notes provide default views, while the Newton only provides default views of subsets of its data store via applications (for instance the Newton notes application provides a default view of "notes" data objects), while WAIS provides no default view at all. A default view is important. It provides a storage structure that users can always fall back on and that can help to prevent the "lost document problem"[2] in search-based systems. The default view typically also provides a convenient way for users to browse new information.

Virtual Directories. Does the system allow virtual collections of information to be created? Organizational constructs like virtual directories allow users to organize data in the way it is needed rather than the way it was created.

Persistent Filtering. If virtual collections of data are allowed, do these collection continue to collect (filter) new information as it is added to the system? Doing so allows users to monitor and categorize incoming information automatically. For instance, Lifestreams allows search results (substreams) to persist, while the Apple Find application does not.

Data Integration. Does the system support its own internal data model or does it also integrate external document types? This point may need further illustration. Guy Friday for instance, uses its own document type, the "nugget." Nuggets can not be exported (without the user explicitly copying its data to another document) nor can Guy Friday incorporate other document types (such as a Microsoft Word document). The Semantic File System, on the other hand, uses transducers to provide an extensible method of handling a variety of document types.

Communication Integration. Is communication integrated into the system? That is, is there a convenient method of exchanging data or documents with others? For instance Lifestreams includes a transfer operation while Guy Friday has no built in method of communication.

Reminders. Does the system allow for the creation of reminders and scheduling information? Barreau and Nardi [BN95] have shown that reminding plays a critical role in typical computer use.

First-class Reminders. If the system has a notion of a reminder, are the reminders first class? That is, can we operate on the reminder like any other document in the system? Are reminders searchable? Can we send reminders to other users? Lifestreams provides all these capabilities while systems like Guy Friday provide reminding that is separate from the data model of the system (e.g., reminders are not nuggets).

[2] The problem of having a document in the system that can't be easily found.

Task Automation. Can tasks be automated by the user? Does the system include an internal scripting language or some other means of automating tasks?

Extrapolation. Can the system be extended to include new functionalities beyond its core capabilities?

Summaries. Does the system have the ability to provide a summary or overview of a data collection? This capability often allows users to quickly assimilate data, draw trends from time-dependent data, or generate overview documents such as monthly reports or billing data.

We now present a feature comparison over the landscape of the systems we have surveyed in Table 8.1. This table is particularly interesting because when analyzing it for common patterns we find the software world in a rather random state. One might expect that the table would neatly partition itself into a small set of similar classes. Instead we find that each system incorporates some aspect of the total landscape of features, but that on a case by case basis, the assignment of features to systems are somewhat random. We believe this makes a strong case for the search for a unified framework that naturally accommodates all features. We note that Lifestreams fulfills this role.

System	Personal Use	Searchable Content	Indexed Content	Transparent Storage	Default Data View	Virtual Directories	Persistent Filtering	Data Integration	Communication Integration	Reminders	First-class Reminders	Task Automation	Extrapolation	Summaries
Apple Find	●					○		●				○		
Contact Manager	●	●							○	●		●		○
Dynamic Queries		●	○			●								○
Emacs	●	●			●	●			●			●	●	○
Enabled Mail	●							●	●	○	○	●	○	
FGP		●	●									○	●	●
Glimpse	●	●	●		○	●		●				○	○	
Guy Friday	●	●	●	●						●		○		
InfoLens	●	○				○		●	●	●	○	●	●	
Memoirs	●	●		●	●	○		●		○				
MIT SFS	●	●	●			●		●				○		
LifeLines	○	●			●									○
Lotus Notes		●	●	●	●	●		●	●			●	●	○
Newton	●	●	●	●	○			●	●	●				
Meeting Maker								○	●	●	○			
Tapestry		●	●		●	●	●		●			●	○	
ToDo List	●	●		○	●					○		●		○
WAIS		●	●			●	○	●				○	○	
Workflow								●	●	●		●	●	
Lifestreams	●	●	●	●	●	●	●	●	●	●	●	●	●	●

Table 8.1: **Comparison features over landscape of systems.** ● indicates the presence of the capability, ○ indicates that the capability is supported in a limited way, or that the capability can, in principle, be supported. An empty field in the table signifies the lack of that feature in the system, or a feature that is not applicable to the system.

Chapter 9

Conclusions

We have shown Lifestreams to be an intriguing and novel system for managing electronic information and events. By providing a common time-based storage structure organized on demand, Lifestreams unifies existing applications and can be extrapolated to new valuable behaviors. Moreover, we have developed a research prototype that has surpassed being a simple "proof of concept" and is now thought of as an indispensable tool by the handful of local research group members that use it. In summary, we have shown that Lifestreams holds the following advantages over current software systems:

- Lifestreams transparently stores information, allowing users to concentrate on the task at hand rather than the name, folder, disk, machine, or network of a particular data item.
- Lifestreams stores information *at the time* it is created and organizes information *in the context* it is needed. This reduces the overhead of creating information, improves recall, and facilitates retrieval.
- Lifestreams is the first general system to treat reminders as first-class entities and to provide a metaphor that naturally accommodates reminding.
- Lifestreams solves the conceptual "data archiving problem" inherent in desktop systems by moving data out of view as it is no longer needed, yet maintaining it for future retrieval.
- Lifestreams provides new opportunity for users to exploit relationships and global patterns that exist in document collections by providing architectural framework for creating executive summaries and overviews.

We believe Lifestreams has a promising future, however the utility of Lifestreams can only be determined through long term study, use, and user acceptance. On this path, there are many avenues for future work. We plan to continue Lifestreams work

at Yale and there has been significant interest from both the research and commercial communities. Already Lifestreams has served as a foundation for three undergraduate projects at Yale. While this dissertation motivated the architecture and demonstrated its usefulness, future work remains in many areas including, but not limited to, improved information retrieval and expert database techniques, scalability, availability, security, user interface design, as well as further study of utility and usability.

Bibliography

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[Aza95] Beth Azar. Meet 'bob': psychology's answer to easy computing. *APA Monitor*, April 1995.

[Bar95] Doug Bartholomew. A better way to work. *Information Week*, pages 32–40, September 1995.

[BF92] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of internet message bodies, June 1992.

[BFJ+95] Martin G. Brown, Jonathan T. Foote, Gareth J. F. Jones, Karen Sparck Jones, and Steve J. Young. Automatic content-based retrieval of broadcast news. In *ACM Multimedia 95 - Electronic Proceedings, San Francisco, CA.*, 1995.

[BH94] B. Bederson and J. Hollan. Pad++: A zooming graphical interface for exploring alternate interface physics. In *ACM UIST*, 1994.

[BN95] Deborah Barreau and Bonnie A. Nardi. Finding and reminding: File organization from the desktop. In *SIGCHI Bulletin*. SIGCHI, July 1995.

[Bor] N.S. Borenstein. *Multimedia Applications development with the Andrew Tool Kit.*

[Bor93a] N. Borenstein. A user agent configuration mechanism for multimedia mail format information, March 1993.

[Bor93b] Nathaniel S. Borenstein. Email with a mind of its own: The safe-tcl language for enabled mail, Nov 1993.

[BR93] Nathaniel Borenstein and Marshall T. Rose. MIME extensions for mail-enabled applications: application/Safe-Tcl and multipart/enabled-mail, Nov 1993.

[Car82] J. M. Carroll. Learning, using and designing filenames and command paradigms. *Behaviour & Info Tech*, 1(4):327–346, 1982.

[CDN88] John P. Chin, Virginia A Diehl, and Kent L. Norman. Development of a tool measuring user satisfaction of the human-computer interface. Technical report, Department of Psychology, University of Maryland, College Park, MD 20742, 1988.

[CFFG96] Nicholas Carriero, Scott Fertig, Eric Freeman, and David Gelernter. Lifestreams: Bigger than Elvis. Technical Report 1098, Yale University Department of Computer Science, March 1996.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, April 1989.

[Col82] I. Cole. Human aspects of office filing: Implications for the electronic office. In *Proceedings of the HUMAN FACTORS SOCIETY—26th ANNUAL MEETING*, 1982.

[Com87] Apple Computer. The knowledge navigator, 1987.

[Com93] Apple Computer. *The Newton Programmer's Guide*. Apple Computer, 1993.

[Coo95] Terry Cook. Do you know where your data are? In *Technology Review*. MIT, January 1995.

[Cor96] Diba Corporation. Vision white paper, 1996.

[Dat86] C. J. Date. *Database Systems*. Addison-Wesley, 1986.

[DL83] S.T. Dumais and T.K. Landauer. Using examples to describe categorizes. *CHI'83*, 1983.

[Dum96] Susan T. Dumais. Personal communication, Jan 1996.

[Eri91] Thomas Erickson. Designing a desktop information system: Observations and issues. *ACM CHI'91*, 1991.

[Eri96] Thomas Erickson. The design and long-term use of a personal electronic notebook: A reflective analysis. *CHI'96*, 1996.

[FE92] William B. Frakes and Ricardo Baeza-Yates (Editors). *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[Flu] Christian Fluhr. Multilingual information retrieval. In *Survey of the State of the Art in Human Language Technology, Varile and Zampolli (Eds.), National Science Foundation and the European Commission.*

[Fre95] Eric Freeman. Lifestreams for the Newton. *PDA Developer*, 3(4):42–45, July/August 1995.

[Gat96] Bill Gates. Keynote address, electronic mail association, Jan 1996.

[Gel91] David Gelernter. *Mirror Worlds*. Oxford University Press, 1991.

[Gel94] David Gelernter. The cyber-road not taken. *The Washington Post*, April 1994.

[GJSO91] David K. Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole. Semantic file systems. In *13th ACM Symposium on Operating Systems Principles*, October 1991.

[GKM95] Burra Gopal, Paul Klark, and Udi Manber. Combining browsing and searching. Technical report, Department of Computer Science, University of Arizona, October 1995.

[GNOT92] David Goldberg, David Nicols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[Hal] Per-Kristian Halvorsen. Document retrieval: Overview. In *Survey of the State of the Art in Human Language Technology, Varile and Zampolli (Eds.), National Science Foundation and the European Commission.*

[Hil95] Ed Hilpert. Human-computer interaction overview. *IBM Personal Systems*, September 1995.

[HN93] B. D. Harper and L. K. Norman. Improving user satisfaction: The questionnaire for user interaction satisfaction version 5.5. In *Proceedings of the 1st Annual Mid-Atlantic Human Factors Conference*, pages 224–228, 1993.

[HS94] M. Holsheimer and A.P.J.M. Sibes. Data mining: The search for knowledge in databases. Technical Report 9406, Cenrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1994.

[Hup96] Susanne Christine Hupfer. *Turingware: An Integrated Approach to Collaborative Computing*. PhD thesis, Yale University, New Haven, Connecticut, 1996.

[Hut95] John Hutchins. Introduction to text summarization. In *Dagstuhl Seminar Report, IBFI, Dagstuhl, 1995*, 1995.

[JD83] William P. Jones and S.T. Dumais. The spatial metaphor: Experimental tests of reference by location versus name. *ACM Transactions on Office Information Systems*, 4(1):43–63, 1983.

[Jon] Karen Sparck Jones. Document retrieval: Summarization. In *Survey of the State of the Art in Human Language Technology, Varile and Zampolli (Eds.), National Science Foundation and the European Commission.*

[Kah91] Brewster Kahle. An information system for corporate users: Wide area information servers. Technical report, Thinking Machines Coporation, April 1991.

[Kap91] Mitchell Kapor. A software design manifesto. *Dr. Dobbs Journal*, 1991.

[Kay90] Alan Kay. User interface: A personal view. In *The Art of Human-Computer Interface Design (Ed.) Brenda Laurel*, 1990.

[KM95] Paul Klark and Udi Manber. Developing a personal internet assistant. In *ED-MEDIA '95 World conference on educational multimedia and hypermedia*, June 1995.

[KM96] Judith L. Klavans and Kathleen McKeown. Domain independent summarization: Project description, 1996.

[Lan88a] M. Lansdale. Memoirs: A personal multimedia information system. *Personal Information Systems: Business Applications, P.J. Thomas (Ed.)*, 1988.

[Lan88b] M. Lansdale. The psychology of personal information management. *Applied Ergonomics*, March 1988.

[LOS93] H. Liao, M. Osada, and Ben Shneiderman. Browsing Unix directories with dynamic queries: An analytical and experimental evaluation. *Proc. Ninth Japanese Symp. Human Interface*, pages 95–98, 1993.

[LS96] Andrew Larratt-Smith. A calendar interface for lifestreams. Technical report, Senior Project, Department of Computer Science, Yale University, May 1996.

[Mal83] Thomas W. Malone. How do people organize their desks? Implications for the design of office information systems. *ACM Transactions on Office Systems*, 1(1):99–112, January 1983.

[Mee94] On Technology, Corp., Cambridge, MA. *Meeting Maker XP Version 2.0 User's Guide*, 1990-1994.

[MGL+87] Thomas W. Malone, Kenneth R. Grant, Kum-Yew Lai, Ramana Rao, and David Rosenblitt. Semistructured messages are surprisingly useful for computer-supported coordination. *ACM Transactions on Office Information Systems*, 5(2):115–131, April 1987.

[MH92] J.T. Mayes and N.V. Hammond. Why is it so hard to learn to use computers? *DTI "Usability Now" Report*, 1992.

[MSW92] Richard Mander, Gitta Salomon, and Yin Yin Wong. The 'Pile' metaphor for supporting casual organization of information. *ACM CHI'92 Proceedings*, pages 627–634, May 1992.

[MW93] Udi Manber and Sun Wu. Glimpse: A tool to search through entire file systems. Technical Report 093-34, Department of Computer Science, The University of Arizona, October 1993.

[Nel90] Theodor Nelson. The right way to think about software design. In *The Art of Human-Computer Interface Design (Ed.) Brenda Laurel*, 1990.

[Nie93] J. Nielsen. *Usability Engineering*. Academic Press, 1993.

[Nor90] Kent L. Norman. Questionnaire for user interaction satisfaction. *Maryland Imagination*, October 1990.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Ove95] Overview. Summarizing text for intelligent communication. In *Dagstuhl Seminar Report, IBFI, Dagstuhl, 1995*, 1995.

[Pem96] Steven Pemberton. Email workshop. In *CHI'96*, Vancouver, BC, 1996.

[Per96] Gary Perlman. *Practical Usability Evaluation, Tutorial Notes*. Conference on Human Factors in Computing Systems, Vancouver, BC, 1996.

[PMR+95] Catherine Plaisant, Brett Milash, Anne Rose, Seth Widoff, and Ben Shneiderman. Lifelines: Visualizing personal histories. Technical Report 787, Human-Computer Interaction Laboratory, Center for Automation Research, Department of Computer Science, University of Maryland, September 1995.

[Por80] M.F. Porter. An algorithm for suffix stripping. *Program*, 14(34):130–137, Jul 1980.

[Shn92] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing Company, 1992.

[Shn94] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, pages 70–77, November 1994.

[SLR84] Praveen Seshadr, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *ACM SIGMOD Conference on Data Management*, 1984.

[Sno90] Richard T. Snodgrass. Temporal databases - status and research directions. *SIGMOD Record*, 19(4):83–89, 1990.

[SSU95] Avi Silberschatz, Mike Stonebraker, and Jeff Ullman. Database research: Achievements and opportunities into the 21st century. Technical report, Report of an NSF Workshop on the Future of Database Systems Research, May 1995.

[Ste91] Richard Stevens. *Distributed Programming*. Addison Wesley, 1991.

[Tuf90] Edward Tufte. *Envisioning Information*. Graphics Press, 1990.

[War95] Blake Ward. *ToDo List: A To Do List Manager*. LandWare, Software for Terra Firma., Budd Lake, NJ, 1995.

[Way95] Peter Wayner. *Agents Unleashed: A public domain look at agent technology*. AP Professional, 1995.

[Wei91] Mark Weiser. The computer for the twenty-first century. *Scientific American*, September 1991.

[WM94] Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical report, Department of Computer Science, The University of Arizona, May 1994.

[WS96] Steve Whittaker and Candace Sidner. Email overload: exploring personal information management of email. *CHI'96*, 1996.

Appendix A

Lifestreams Primitives

```
func extract(int docid, char attr)
begin
    document DS;

    rd ("document", docid, ?  DS);
    return ds_extract(DS, attr);
end

proc replace(int docid, char attr, poly val)
begin
     document DS;

     in ("document", docid, ?  DS);
     ds_replace (DS, attr, val);
     out ("document", docid, DS);
end
```

Figure A.1: The complete stream primitives.

```
proc read(int sid, int docindex, document DS)
begin
     int docid;

     rd("streamdoc", sid, docindex, ? docid);
     rd("document", docid, ? DS);
     return DS;
end

proc write(int sid, int docindex, document DS)
begin
     document oldDS;
     int created;

     rd("streamdoc", sid, docindex, ? docid);
     in("document", docid, ? oldDS);
     created := ds_extract(oldDS, "created");

     ds_replace(DS, "created", created);
     out("document", docid, ? DS);
end

proc append(int sid, document DS)
begin
     int docid, docindex, filterindex, valid;
     int agentnum, i;
     char agentcode;

     valid := (extract(sid, docid, "created") >= now)
     if (valid) then
     begin
       in("documenthead", ? docid);
       out("documenthead", docid + 1);
       in("stream", sid, ? docindex, ? subhead, ? agentnum);
       out("stream", sid, docindex + 1, subhead, agentnum);
       out("document", docid, DS);
       out("streamdoc", sid, docindex, docid);
       for(i := 0; i < agentnum; i++)
       begin
          rd("agent", i, ? agentcode);
          eval(agentcode(sid, docindex));
       end
       return docindex;
     end
end
```

Figure A.2: The complete stream primitives (cont).

```
proc filter(int sid, int pid, char query)
begin
     int docnum, subnum;
     in("stream", sid, ? docnum, ? subnum);
     out("stream", sid, docnum, subnum + 1);
     out("substream", sid, subnum, pid, query);
     return subnum;
end

proc retrieve(int sid, int subid)
begin
     char query;
     int docnum, subnum, doc, pid;
     list docs, substream := ();

     rd("substream", sid, subid, ? pid, ? query);
     rd("stream", sid, ? docnum, ? subnum);
     if (pid == 0)
       docs := iota(docnum);
     else
       docs := retrieve(sid, pid);
     foreach doc in docs
        if (match(sid, doc, query))
          substream := cons (substream, doc);
     return sort(sid, substream, "created");
end

proc add_agent(int sid, char agent)
begin
     int dochead, subhead, agentid;
     in("stream", sid, ? dochead, ? subhead, ? agentid);
     out("stream", sid, dochead, subhead, agentid + 1);
     out("agent", sid, agentid, agent);
     return agentid;

lifestreams := filter(sid, 0, "*");
```

Figure A.3: The complete stream primitives (cont).

Appendix B

User Questionnaire

```
Lifestreams User Questionnaire


 Name of hardware: Unix version
 Name of software: Lifestreams

 Identification number:  ________________________
 Age:  ______
 Sex:   __  male  __ female


PART 1:  Type of System to be Rated


 1.1   How  long have you worked on this system?

     __  less than 1 hour             __  6 months to less than 1 year
     __  1 hour to less than 1 day    __  1 year to less than 2 years
     __  1 day to less than 1 week    __  2 years to less than 3 years
     __  1 week to less than 1 month     __  3 years or more
     __  1 month to less than 6 months

 1.2   On the average, how much time do you spend per week on this system?

     __  less than one hour                __  4 to less than 10 hours
     __  one to less than 4 hours         __  over 10 hours

PART 2:  Past Experience

 2.1   How many different types of computer systems (e. g., main frames and
       personal computers) have you worked with?
```

```
__  none      __  3-4
__  1         __  5-6
__  2         __  more than 6
```

PART 3: Overall User Reactions

Please circle the numbers which most appropriately reflect your impressions about using this computer system. Not Applicable = NA.

3.1 Overall reactions to the system:

terrible wonderful
1 2 3 4 5 6 7 8 9 NA

3.2 frustrating satisfying
1 2 3 4 5 6 7 8 9 NA

3.3 dull stimulating
1 2 3 4 5 6 7 8 9 NA

3.4 difficult easy
1 2 3 4 5 6 7 8 9 NA

3.5 inadequate power adequate power
1 2 3 4 5 6 7 8 9 NA

3.6 rigid flexible
1 2 3 4 5 6 7 8 9 NA

PART 4: Screen

4.3 Were the screen layouts helpful?

never always
1 2 3 4 5 6 7 8 9 NA

4.3.1 Amount of information that can be displayed on screen

inadequate adequate
1 2 3 4 5 6 7 8 9 NA

4.3.2 Arrangement of information on screen

illogical logical

1 2 3 4 5 6 7 8 9 NA

Please leave any comments that you have about the screens here:

PART 6: Learning

6.1 Learning to operate the system

difficult easy

1 2 3 4 5 6 7 8 9 NA

6.1.1 Getting started

difficult easy

1 2 3 4 5 6 7 8 9 NA

6.1.2 Learning advanced features

difficult easy

1 2 3 4 5 6 7 8 9 NA

6.1.3 Time to learn to use the system

slow fast

1 2 3 4 5 6 7 8 9 NA

Please leave any comments that you have about learning the system here:

PART 7: System Capabilities

7.1 System speed

too slow fast enough

```
                                    1  2  3  4  5  6  7  8  9       NA

    7.1.1   Response time for most operations
                              too slow                    fast enough
                                    1  2  3  4  5  6  7  8  9       NA

    7.1.2   Rate information is displayed
                              too slow                    fast enough
                                    1  2  3  4  5  6  7  8  9       NA

7.2   How reliable is the system?
                              unreliable                   reliable
                                    1  2  3  4  5  6  7  8  9       NA

    7.2.1   Operations are
                              undependable              dependable
                                    1  2  3  4  5  6  7  8  9       NA

    7.2.2   System failures occur
                              frequently                  seldom
                                    1  2  3  4  5  6  7  8  9       NA


7.5   Are the needs of both experienced and inexperienced users taken
      into consideration?
                              never                              always
                                    1  2  3  4  5  6  7  8  9       NA

    7.5.1   Novices can accomplish tasks knowing only a few commands
                              with difficulty               easily
                                    1  2  3  4  5  6  7  8  9       NA

    7.5.2   Experts can use features/shortcuts
                              with difficulty               easily
                                    1  2  3  4  5  6  7  8  9       NA

Please leave any comments that you have about the system capabilities here:
_____________________________________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
```

Part 8: General Questions

What was your initial reaction to Lifestreams upon hearing the concept, seeing the system or first using the system?

What features of Lifestreams do you find most useful when compared with your previous computing environment(s)?

What features of your previous computing environment do you miss when using Lifestreams?

Do you find any aspect of Lifestreams difficult or confusing?

Does Lifestreams make any aspect of managing information less (or more) confusing (say, as compared to the way you normally do things)?

Does Lifestreams in any way change the way you thought about using your computer or managing your information? If so, how?

Did you quickly understand the "receding stream" user interface? Does it make sense to you?

---

[a] Was it an effective interface to your stream of documents? That is, does it allow you to carry out the operations you needed to?

---

[b] Do the animations help you to understand the effect of the various operations? For instance creating a new document, the arrival of a new document, sending out a document, creating a substream?

Do you like the fact that all your documents (email, TeX documents, pictures, bookmarks, etc.) were stored in the same stream? Or do you find it easier to maintain them separately in your file system, email application, etc.?

Do you find chronology (the time-based stream) a useful method of storing and managing information?

---

[a] Does chronology allow you to concentrate on current information while older information was moved to the background? Or would you prefer some other default organization?

---

[b] Does chronology help you locate older information (say, because you remember the approximate time when that document was created)?

---

[c] Do you want a delete button in your lifestream? If so, why?

Do you like that fact that you don't have to file and name new documents? Or do you prefer to name them and store them in a directory?

Do you find substreams useful for locating information?

---

[a] What about for maintaining collections of information over time?

---

[b] Were you usually able to find what you were looking for? If not, was the system frustrating in this respect?

---

[c] Do you typically keep substreams around for future use, or do you remove them soon after they are created? Or do you do a bit of both?

Do you make use of the reminding functionality of the future part of the stream?

---

[a] Does this "metaphor" for reminding make sense to you?

Do you use any "squishes"? Which ones? How do they help you?

What features or changes would you like to see in Lifestreams?

If Lifestreams were a robust and supported piece of software would you
continue using it?

